DAA/LANGLEY
NSG-1471
IN-62-CR
76288
P-246

1987 Mid–Year Report

EOS: A Project to Investigate the Design and Construction of Real–Time Distributed Embedded Operating Systems.

```
(NASA-CR-180971)  EOS: A PROJECT TO                    N87-26577
INVESTIGATE THE DESIGN AND CONSTRUCTION OF
REAL-TIME DISTRIBUTED EMBEDDED OPERATING
SYSTEMS Mid-Year Report, 1987  (Illinois               Unclas
Univ.)  246 p  Avail: NTIS HC  A11/MF  A01    G3/62   0076288
```

Principal Investigator: R. H. Campbell.

Research Assistants:
Ray B. Essick,
Gary Johnston,
Kevin Kenny,
Vince Russo.

Software Systems Research Group

University of Illinois at Urbana–Champaign
Department of Computer Science
1304 West Springfield Avenue
Urbana, Illinois 61801–2987
(217) 333–0215

TABLE OF CONTENTS

ABSTRACT:

Project EOS is studying the problems of building adaptable real–time embedded operating systems for the scientific missions of NASA. Choices, a *Class Hierarchical Open Interface for Custom Embedded Systems*, is an operating system designed and built by Project EOS to address the following specific issues: the software architecture for **adaptable embedded parallel operating systems**, the achievement of **high–performance** and **real–time operation**, the simplification of **interprocess communications**, the **isolation of operating system mechanisms** from one another, and the **separation of mechanisms from policy decisions**. Choices is written in C++ and runs on a ten processor Encore Multimax. The system is intended for use in constructing specialized computer applications and research on advanced operating system features including fault–tolerance and parallelism.

One of the applications made possible by our research is a software system that allows workstation applications to be **closely integrated** with software running on specialized computers like a supercomputer or supermini. CLASP is a mechanism that allows the **virtual memory space** of a workstation to be **shared** with a high–performance computer. CLASP implements a **cross–architecture procedure call** that allows an application on a workstation transparently to invoke procedures on the high–performance machine. The method allows existing software packages to be decomposed **without change** onto a compatible workstation supercomputer or supermini computer pair. Ray Essick's Ph.D. thesis documents this work.

1. Introduction.

Project EOS is investigating the design and construction of embedded real–time systems for applications in NASA's aerospace programs. The results of our study in previous years is documented in the bibliography in Appendix A. In the first six months of the current grant period, we built a prototype adaptable embedded real–time operating system for parallel computers called *Choices*, designed and built *CLASP*, a mechanism that uses virtual memory to implement a flexible remote procedure call, and, to satisfy several requests for previous Project EOS work, created a new release of the Path Pascal compiler for Berkeley UNIX™ BSD 4.3. An interface compiler for Choices has been designed, but is not yet implemented.

2. Choices

Choices is an experimental real–time embedded operating system for parallel and distributed computer systems in aerospace applications. The initial prototype has been built on a ten processor Encore Multimax. The system is designed to support:

- the object–oriented organization of user applications,
- applications requiring custom designed operating systems,
- diverse hardware architectures (both networked computers and shared memory multiprocessors),
- parallel computation where performance is an issue,
- persistent objects,
- protection,
- real–time operation of applications,
- research and applications requiring specialized operating system functions.

The design of the operating system reflects an object–oriented approach. The code is organized to meet a number of objectives:

- The software is to be placed in the public domain.
- The software is organized as a hierarchy of classes written in C++. C++ is implemented, currently, as a preprocessor for C.
- Classes separate operating system mechanism from policy and allow reuse of modules.
- The classes used in Choices may be specialized or modified to create new operating system features without jeopardizing the architectural integrity of the system and

UNIX is a Registered Trademark of AT&T.

should encourage advanced operating system research.

- The systems programming language C++ has not been extended or modified; all process, exception and communication mechanisms are written using classes. This encourages portability.
- Hardware and application specific features are encapsulated in classes and separated from device independent and application independent code.
- Many operating system services execute in application space, reducing the size of the Choices kernel.
- The system and its applications use UNIX loader formats and can be built under UNIX.

The system is intended to support future investigations of real–time software organization, fault–tolerance, networked computers, and load balancing. Much of the design of Choices can be translated into Ada™ or other non object–oriented languages. This would permit "high–quality production" implementations of the code. However, for the purposes of this research, C++ has been excellent. The subclassing and generic functions of C++ have many advantages in prototyping and maintaining code consistency. C++ produces good, fast code, it aids and speeds recompiling and software reuse, and it has been ported to a large number of machines. It is available at a minimal cost for a license to research organizations. The source of the compiler, linkers, and other utilities are available. At this point in time, C++ has many advantages for our experimental operating system work.

Choices is discussed further in Appendix B. The prototype code for Choices (as of May 21, 1987) can be found in Appendix C.

3. CLASP

CLASP, provides a new implementation of the traditional process model. It allows portions of the process to execute on the most appropriate processor architecture. CLASP isolates a practical level of homogeneity necessary to implement this sharing; it also mitigates dissimilarities between the processor architectures — such as register sets and stack frame formats.

CLASP makes the address space of a single process available to heterogeneous CPUs with potentially different instruction sets and performance characteristics. Where other approaches have concentrated on enhancing addressing to include the concept of remote addresses, CLASP makes a single address space accessible to multiple heterogeneous CPUs. A novel aspect of the CLASP architecture is the inclusion of instructions for different processor architectures within the same address space. The CLASP system introduces a new construct, the Cross Architecture Procedure Call, to transfer a process's control thread between CPUs. The Cross Architecture Procedure Call — or

Ada is a Registered Trademark of the Department of Defense.

CAPC — uses each CPU's subroutine call and return instructions to implement control transfers between CPUs. This control transfer mechanism and the shared address space make CAPCs more transparent than Remote Procedure Calls (RPCs), which require special stub routines and system calls to implement control transfers between CPUs.

To use the CLASP architecture, a special CLASP loader links separately compiled routines. The CLASP loader recognizes the different object formats for various processor architectures and resolves the cross–architecture references. It provides the operating system kernel with the information necessary to detect control transfers (e.g., procedure and function calls) that cross architecture boundaries. Routines to execute on specific architectures are compiled for those architectures. Some frequently called routines (e.g., sqrt()) are replicated. Duplicate copies of these routines, each compiled for a different architecture, are loaded into the executable file. Calls to any of these routines can be directed to the local instance of that routine, saving the network overhead of a remote call. This replication is a loader operation.

CLASP subroutine libraries may contain routines for several architectures. Specific routines within a library can be compiled for the most appropriate processor architectures. A library of subroutines to manipulate large arrays may contain code for several architectures; for example, routines that manipulate the array may be compiled for high performance vector architectures, such as that provided by the Convex C–1. Other routines in the library, which do not perform large calculations, may be compiled for the workstation architecture.[1]

Trees, lists, and other pointer–based data structures are difficult and sometimes impractical to implement in distributed computing models without a shared address space. The SUN Remote Procedure Call dereferences pointers to pass individual elements of a pointer–based structure. Pointer dereferencing is adequate for situations where single structures are passed by pointer instead of value. Nelson advocates the use of subroutines to encapsulate access to pointer–based structures. This approach implies changing (or deliberately designing) the applications program to encapsulate accesses to these structures. The CLASP software architecture addresses this problem by ensuring that the context for a pointer (i.e., its address space) is in effect on the remote processor. Applications may use pointers as handles to objects and for true pointer–based structures without concern about where a procedure is implemented.

Many RPC implementations package the entire argument list and send it to the remote host. Datagram based RPC implementations send the entire argument list to the server in a single packet. Therefore, the argument list must be small enough to fit into a single packet. Some implementations provide larger argument lists by supporting stream based connections. CLASP supports arbitrary sized argument lists. CLASP uses

[1] These routines also might be compiled for both client and server architectures. Calls to the replicated routine can be directed to the local instance of that routine and avoid the overhead of a network transaction. As was pointed out in the above paragraph, the loader performs this replication and resolves references to send most calls to that routine to a local instance of the routine.

demand paging to move arguments and data to the server only on request for access by the remote procedure. As an example, binary searches through large sorted arrays can be efficient because only the accessed portions of the array are transferred to the remote processor. Pages, once transferred to the server, remain on the server until they are required by the client processor. By leaving the pages on the server, most data eventually becomes local to a particular computer system. Pages used only by the client remain on the client; pages used only by the server will be transferred to and remain on the server. Pages of data used by both processors will migrate between hosts as needed.

Although CLASP appears to be an approach to distributed computing, it is actually an extension of the traditional single–system model onto a new underlying implementation for greater performance and ease of use. CLASP mimics a single processor model but allows the most appropriate CPU to process appropriate parts of the problem. The application program is neither restructured nor recompiled. The choice of which processor performs a specific routine affects only the processing rate for that procedure. The choice does not alter the semantics for that procedure nor its interactions with other procedures in the address space.

CLASP has been implemented between SUN 3 systems under UNIX. In the next year, CLASP will be implemented in Choices. Appendix D contains Ray Essick's Ph.D. thesis on CLASP which details the work done in the last six months.

4. Path Pascal Release

A new release of Path Pascal for Berkeley UNIX BSD 4.3 was made this Spring. The release was prompted by a number of requests for Path Pascal for SUN workstations. The new release corrected a number of bugs in the BSD 4.2 version. The new release of Path Pascal has been used for the operating system class at the University of Illinois. The new release has been distributed to five sites including the Electrical Engineering Department at Cornell where it was used in a network simulation class. The new release can be obtained on request from Professor Campbell.

5. The Choices Interface Compiler

In Choices, there are a large number of operations that may be conceived of as being ***wrapped around*** user–described operations. For example, a call to a persistent object involves remapping the address space as part of the call and (possibly) again as part of the return. Parameter transmission across this interface in some cases (for example, when the object is remote) is not straightforward.

There are many other examples of this type of "wrapped" interface. The implementor of an object may well want to impose a synchronization discipline upon its callers, as in the Open Path Expressions used with Path Pascal. This is also best described as actions to be taken before and after executing the called procedure; in this case, the actions are the appropriate operations on synchronization objects (such as semaphores or events).

A remote procedure call is a more complex example of the same sort of "wrapped" interface. The action that takes place before a call is to prepare a description of the called procedure and the parameters for transmission to the server; the action on return is to get the description of return value and result parameters from the server and format them correctly for the caller again. The server's logic to handle the specific remote procedure call is also a "wrapped" interface with a similar flavor.

There are many other examples, such as preserving atomicity, journalization of input and output letters in a transaction processing system, and establishing commitment points in a database manager, that are all examples of "wrapped" interfaces. The common factors for all of these are:

1 The procedure to be executed consists of taking some action, eventually calling another procedure *with the same calling sequence as the first,* and then taking some other action before returning.

2 The procedure can be generated from a knowledge of its calling sequence, without needing to be adapted to the specific application. For example, all calls to an operating system kernel can be translated to the corresponding `trap` operations without knowledge of the function of any particular call.

If both of these requirements are met, the procedure is a candidate for *interface compiling.*

The Interface Compiler

The interface compiler is a program that has, as input, a description of the object or objects to be adapted (a C++ `#include` file), and a description of how to generate the type of interface required (kernel calls, remote procedure calls, or whatever). It produces any number of source files as output; these files give the code needed to implement the interface. Multiple source files are often required because the interface may need to be compiled into multiple object modules; for instance, the server and client ends of a remote procedure call.

6. Summary

Based on our prior research and in cooperation with Ed Foudriat, Project EOS has built Choices, a prototype experimental embedded real–time system for parallel and network computer architectures. The system has been implemented on a 10 node Encore Multimax. The organization of the software is novel and demonstrates that operating systems can be constructed using an object–oriented methodology. The current system includes parallel execution of Threads on the Multimax, implementation of Spaces, and handling of exceptions and interrupts. Future work will extend this operating system with servers, real–time features, and networking.

Many of the networking aspects of Choices has been prototyped in the CLASP system. This architecture and software system allows a virtual memory to be shared between several processors. An implementation of CLASP was built for the SUN 3 workstation.

A new release of Path Pascal was created for Berkeley UNIX BSD 4.3 because of popular demand.

Finally, progress is being made in the design and development of a Choices interface compiler that will aid the construction of network servers, debugging tools, and other utilities and services.

APPENDIX A

Project EOS Bibliography

1. Campbell, R. H., K. Horton, and G. G. Belford, *Simulations of a Fault-Tolerant Deadline Mechanism,* **Digest of Papers FTCS–9: Ninth Annual International Symposium on Fault–Tolerant Computing,** Madison WI, June, 1979, 95–102.
2. Campbell, R. H. and R. B. Kolstad, *Path Expressions in Pascal,* **Proceedings of the Fourth International Conference on Software Engineering,** Munich, September 17–19, 1979, 212–219.
3. Campbell, R. H. and R. B. Kolstad, *Practical Applications of Path Expressions to Systems Programming,* **ACM79,** Detroit, 1979, 81–87.
4. Campbell, R. H. and R. B. Kolstad, *An Overview of Path Pascal's Design,* **Sigplan Notices,** Vol. 15, No. 9, pp. 13–14, September, 1980.
5. Kolstad, R. B. and R. H. Campbell, *Path Pascal User Manual,* **Sigplan Notices,** Vol. 15, No. 9, pp. 15–24, September, 1980.
6. Kolstad, R. B. and R. H. Campbell, *Directions for User Defined Communication for Distributed Software,* **Proceedings of The International Conference on Parallel Processing,** IEEE 80CH1569-3, pp. 188–189, Boyne MI, August 26–29, 1980.
7. Wei, A. Y., K. Hiraishi, R. Cheng, R. H. Campbell, *Application of the Fault–Tolerant Deadline Mechanism to a Satellite On–Board Computer System,* **Digest of Papers FTCS–10: Tenth International Symposium on Fault–Tolerant Computing,** Kyoto Japan, October 1980.
8. Kolstad, Robert Bruce. "Distributed Path Pascal: A Language for Programming Coupled Systems," *Ph.D. Thesis,* Department of Computer Science Technical Report #1136, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1983.
9. Liestman, A. and R. H. Campbell, *A Fault Tolerant Scheduling Problem,* **Digest of Papers FTCS–13: Thirteenth Annual International Symposium on Fault–Tolerant Computing,** Milano Italy, June 1983.
10. Schmidt, George Joseph. "The Recoverable Object as a Means of Software Fault Tolerance," *MS Thesis,* Department of Computer Science, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1983.
11. Campbell, R. H., *Distributed Path Pascal,* **In Distributed Computing Systems,** (Editor Y. Paker and J.–P. Verjus), Academic Press, 1983, pp. 191–224.
12. Campbell, R. H. and T. Anderson, *Practical Fault Tolerant Software for Asynchronous Systems,* **SAFECOMP 83, Third International IFAC Workshop on Achieving Safe Real–time Computer Systems,** Pergamon Press, Oxford, England, 1983.
13. Jalote, Pankaj and Roy H. Campbell, "Fault Tolerance Using Communicating Sequential Processes," In: *Digest of Papers, Fourteenth International Conference on Fault Tolerant Computing, IEEE FTCS14* (June 1984) pp. 347–352.

14. Campbell, Roy H., Jeff Donnelly, Raymond B. Essick, Judith Grass, Dirk Grunwald, Pankaj Jalote and David A. McNabb. "The Embedded Operating System Project," 1984 Mid-Year Report, NASA GRANT NSG 1471, Software Systems Research Group, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1984.

15. McKendry, M. S., and R. H. Campbell, *A Mechanism for Implementing Language Support in High-Level Languages,* **Transactions on Software Engineering,** Vol. SE-10, No. 3, May 1984, pp.227-236.

16. Mickunas, M. D., Pankaj Jalote and Roy H. Campbell. "The Delay/Re-Read protocol for Concurrency Control," In: *Proceedings, First International Conference on Data Engineering.* IEEE, Los Angles, California, 1984.

17. Jalote P. and R. H. Campbell, "Atomic Actions in Concurrent Systems," Proceedings of the *5th International Conference on Distributed Computing Systems,* Denver, May 1985.

18. Grunwald, D. C., "An Implementation of Path Pascal," *MS Thesis,* Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.

19. Jalote P., "Atomic Actions in Concurrent Systems," *Ph.D. Thesis,* Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.

20. Campbell, Roy H., Raymond B. Essick, Judith Grass, Dirk Grunwald, Pankaj Jalote, Kevin Kenny and David A. McNabb. "The Embedded Operating System Project," 1985 Mid-Year Report, NASA Grant NSG 1471, Software Systems Research Group, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.

21. Jalote P. and R. H. Campbell, *Atomic Actions for Fault-Tolerance using CSP,* **IEEE Transactions on Software Engineering, Special Issue on Software Reliability - Part II,** Vol. SE-12, No. 1., January 1986.

22. Grass, J. E., *Mediators,* Ph.D. Thesis, Technical Report, UIUCDCS-R-86-1266, 1986.

23. Grass, J. E. and R. H. Campbell, *Mediators: A Synchronization Mechanism,* **Proc. of Sixth International Distributed Computing Systems,** IEEE, Cambridge, Mass., May 19-23, 1986, pp. 468-477.

24. Campbell R. H. and B. Randell, **Error Recovery in Asynchronous Systems, IEEE Transactions on Software Engineering, August,** 1986.

25. Liestman, A. and R. H. Campbell, *A Fault Tolerant Scheduling Problem,* **IEEE Transactions on Software Engineering,** November 1986.

26. V. Russo, *LINK,* M.S. Thesis, Technical Report, Department of Computer Science, University of Illinois, Urbana, IL 61801, 1987.

26. Essick, R. B., *The Cross Architecture Procedure Call,* Ph.D. Thesis, Technical Report UIUCDCS–R–87–1340, Department of Computer Science, University of Illinois, Urbana, May 1987.

27. Campbell, R. H., G. Johnston and V. Russo, *Choice,* Technical Report, University of Illinois, Urbana, May 1987.

APPENDIX B

Choices

Roy Campbell, Gary Johnston, Vincent Russo

Choices
(Class Hierarchical Open Interface for Custom Embedded Systems[1])

Roy Campbell, Gary Johnston, Vincent Russo

University of Illinois at Urbana-Champaign
Department of Computer Science, 1304 W. Springfield Ave., Urbana, IL 61801-2987

1. Introduction

This paper describes the design for an operating system family called Choices being built for the Embedded Operating System (EOS) project at the University of Illinois at Urbana-Champaign. Choices embodies the notion of *customized* operating systems that are tailored for particular hardware configurations and for particular applications. Within one large computing system, many different specialized application servers may be integrated to form a general purpose computing environment. We have implemented a Choices Kernel on an Encore Multimax.

Choices, a *Class Hierarchical Open Interface for Custom Embedded Systems,* provides a foundation upon which to construct sophisticated scientific and experimental software. Unlike more conventional operating systems, Choices is intended to exploit very large multi-processors interconnected by shared memory or high-speed networks. Uses include applications where high-performance is essential like data reduction or real-time control. It provides a set of software classes that may be used to build specialized software components for particular applications. Choices uses a class hierarchy and inheritance to represent the notion of a family of operating systems and to allow the proper abstraction for deriving and building new instances of a Choices system. At the basis of the class hierarchy are multiprocessing and communication objects that unite diverse specialized instances of the operating system in particular computing environments.

The operating system was developed as a result of studying the problems of building adaptive real-time embedded operating systems for the scientific missions of NASA. Major design objectives are to facilitate the construction of specialized computer systems, to allow the study of advanced operating system features, and to support parallelism on shared memory and networked multiprocessor machines. Example specialized computer systems include support for robotics applications, network controllers, aerospace applications, high-performance numerical computations, parallel language processor servers for IFP[13], Prolog, Smalltalk, and reconfigurable systems. Examples of advanced operating system features include fault-tolerance in asynchronous systems, real-time fault-tolerant features, load balancing and coordination of very large numbers of processes, atomic transactions and protection. Example hardware architectures include shared memory multiprocessors like the Encore Multimax and networked computers like the Intel Hypercube.

Choices was designed to address the following specific issues: the software architecture for *parallel operating systems*; the achievement of *high-performance* and *real-time operation*; the simplification and *improved performance of interprocess communications*; the *isolation of mechanisms* from one another and the *separation of mechanisms from policy decisions.*

[1] This work was supported in part by NASA under grant no. NSG1471 and by AT&T METRONET.

Of particular concern during the development of the system, was whether the class hierarchical approach would support the construction of entire operating systems. C++ was chosen because it supported classes while imposing negligible performance overhead at run-time. In particular, we decided to construct all parallel and synchronization features using C++ classes rather than by introducing new language primitives. Thus Choices is also a study of the adequacy of class hierarchies to abstract and support parallelism and other operating system concepts and to allow specializations of classes that facilitate efficient support for applications.

Fortunately for the designers of Choices, there has been a lot of operating system development that is directly applicable to our goals. However, this development work often produced implementations buried within the bowels of large, successful operating systems. Abstracting the ideas from many different systems and reorganizing them into Choices has been a major concern of our design team.

Choices has been influenced considerably by UNIX™ and MULTICS. Indeed, many of the standard UNIX system compilers, linkers and utility programs have been used to produce the Choices software. However, to structure Choices to allow multiple processes running simultaneously on a multiprocessor with a high degree of parallelism and communication, we have had to abandon the UNIX organization of the kernel. Similarly, the UNIX process supports the sequential execution of a program running within its own address space. To support real-time and high-performance applications, we have opted for a lightweight process. Multiple lightweight processes can run on multiple processors within the same virtual address space. Communication performance in UNIX is limited by coroutining within the kernel and by copying information into and out of user space. In Choices, we have attempted to eliminate these bottlenecks.

The open architecture of Choices is influenced by the ideas used to build CEDAR [16]. The notion of a lightweight process is very similar in Choices and CEDAR, although in Choices it is provided through a class abstraction and is not built into the systems language. The Choices notion of a lightweight process may be specialized through the subclassing mechanism and this is used in the software to distinguish user and system processes. Choices permits a virtual address space to be shared by multiple processors. It offers concurrent applications protection from one another and hence is, in CEDAR terminology, a closed operating system. However, user created operating system policies and mechanisms (like a file system) are provided by the open interface of Choices that is supported through the notion of persistent objects. CEDAR is not completely built as an object-oriented system although the MESA language is oriented towards encapsulated data structures which influences the organization of the system.

Many current operating system designs address the problem of distributed computing. Choices owes several of these systems many of its ideas, but the support of applications on parallel processors has caused us to implement these ideas in different ways. Many distributed/multiprocessor UNIXs (UNIX United [3], LOCUS [10], Mach [1], RFS [13], RIDE [9], NFS [18], Encore Multimax UNIX (UMAX) [7], Sequent Balance 8000 UNIX [14]) still impose UNIX limitations on the parallelism and performance of applications. Multiprogramming on a cached multiprocessor can have undesirable side effects in the form of additional caching and cache flushing overhead. Message-oriented kernels like the V System kernel [6], Accent, Amoeba [17], and Micros [19] build specific communication schemes into the lowest structures in the kernel, restricting the possibilities of specializing kernel features to take advantage of communication patterns of the application or communication mechanisms of the hardware. For example,

™ UNIX is a Registered Trademark of AT&T.

systems implement a few ways of providing "virtual" messages like "fetch on access." However, these systems are not easy to adapt to support other approaches like "send on write", "send on execute", and "remote procedure call on execute." Many systems suffer overhead from copying messages into and out of virtual memory. Cached systems may pay a double overhead.

Real-time interrupts and global multiprocessor interrupts pose organizational problems in traditional operating system architectures like UNIX. Most operating systems do not include parallel programming primitives (for example, the parallel creation of parallel processes), nor can they be built easily out of the primitives that exist in such systems. Error recovery is difficult to provide in current operating system architectures when used for parallel processing without restricting parallelism because atomicity constraints cannot be imposed easily.

The Clouds operating system [2] includes many concepts that have been useful in developing Choices. In particular, its notion of a user process accessing a user object is similar to processes accessing persistent objects in Choices. Choices differs in not supplying a kernel level atomic transaction.

One of the goals of Choices is to permit the custom design of operating systems for specific hardware and applications. General purpose operating systems employ delayed bindings within their architectures to provide flexibility. Examples include communication schemes, file systems and additional kernel code to handle different architectures and configurations. Choices, on the other hand, is aimed at providing the smallest operating system that will support a particular application on a particular hardware. Where several applications need to coexist within the same computing system, Choices allows these applications to each run on their own custom-built Choices operating system. Any communication required between the applications is supported by common Choices primitives and shared persistent objects.

The design of Choices is based upon several assumptions:

- Embedded, real-time, and server computing services will be provided by large numbers of fast processors connected together by shared memory or by a fast network.
- A computational facility is multitasked (it supports several concurrent applications), where each task may use multiple processors.
- Processes in an application have a high degree of communication.
- Each application may need to intercommunicate with other applications. Applications communicate less frequently than lightweight processes within a particular application.
- Communication overheads are small but significant.
- Even though hardware technology will provide large multiprocessors with very fast processors, performance of the applications will remain a critical issue.
- Small lightweight operating systems are desirable in real-time and high-performance applications.
- Processors within a multiprocessor may be dynamically partitioned to execute different applications.
- Each application may need basic support and specialized support from the operating system.
- The hardware will support very large virtual address spaces.

Choices is designed to support specialized applications like embedded real-time systems, numerical programs and specialized computing environments like FP or parallel logic programs. A Choices system could be embedded as a node within a network of workstations.

In the subsequent sections, we discuss the class hierarchical organization of Choices and the various classes we have built to implement virtual memory, the concept of process, the notion of a persistent object and exception handling.

2. The Choices Class Hierarchy Model

Several problems emerge when designing an extensible family of operating systems where each member can be specialized or customized for a particular application or hardware configuration. Each module within the system may have many different versions tailored for each different member of the family of operating systems. However, since the different versions of a module for different machines or applications all perform a similar function, large portions of different versions of a module will be identical. Customizing an operating system for a new application requires access to particular aspects of the code that may reside in many different modules.

A class hierarchy provides an ideal solution to these problems. Particular instances of classes in the hierarchy are chosen and combined to produce a customized operating system for a specific architecture and application. Class inheritance provides for code re–use and enforcement of common interfaces. Customization of the operating system for new applications is guided and aided by the structure induced upon the system by the class hierarchy.

A class hierarchy gives more than ease of customization. It also gives us a conceptual view of how portions of an operating system interrelate. It is easier to understand and more flexible than traditional layered approaches to operating system design. A class hierarchy allows conceptual "chunking" of knowledge about portions of a system by learning the function of parent classes and inferring functionality about subclasses. Traditional layered approaches conceptually group large sections of functionality into a layer, but the interrelations of the layers are often complex and poorly understood. Also changing a piece of a layer is in no way facilitated by the layering. However, in a well designed Class Hierarchical model only the top few classes would need to be mastered to achieve a good overall view of the system. Class derivation gives a method to change specific parts without adversely effecting the whole structure.

The Choices support for applications is divided into two portions. The *Germ* is a set of classes that encapsulates the major hardware dependencies of Choices and provides an "idealized" hardware architecture to the rest of the classes in the hierarchy. It provides the *mechanisms* for managing and maintaining the physical resources of the computer. A *Kernel* is a collection of classes that supports the execution of applications and implements resource allocation *policies* using the Germ mechanisms.

Individual customized systems will consist of derived classes from the Germ classes defined by Choices appropriate for the particular hardware of the system, plus the specifically tailored Kernel classes the system builder desires. Once this hierarchy is laid down, individual applications that run on top of the new Kernel can further augment the class hierarchy with their own classes.

In the following sections, we will describe some of the classes that constitute Choices. The first set of classes we will discuss provides an abstraction for physical and virtual memory.

3. Stores and Spaces

Stores and *Spaces* are classes of objects which the Choices Germ provides for memory management. A Store object encapsulates the management of *physical* memory. An instance of a Store manages a *range* of contiguous *physical* memory addresses. Operations are provided for

Store instantiation, Store destruction, page allocation, and page deallocation. One application of multiple Store objects is to manage memories with different properties, for example, local memory, shared memory within a multiprocessor, or global memory shared between multiprocessors.

A Space object encapsulates the management of *virtual memory*. An instance of a Space manages a *range* of contiguous *virtual* memory addresses. Operations are provided for Space instantiation, Space destruction, allocation and deallocation of page table entries, changing protection flags on page table entries, mapping a page table entry to a physical page of memory within a Store, and mapping virtual memory addressing faults on specific page table entries to appropriate exception handlers (see the section on Exception Handling in Choices.)

Many non–overlapping Spaces may be mapped into the virtual address range of a processor at any one time. The aggregate of the Spaces addressable in a processors virtual memory is represented and managed by an instance of the *Universe* class.

Spaces implement protection of the virtual memory they contain by means of the available virtual memory hardware protection mechanisms. Protection ensures that a process can only access a Space according to the access rights it possesses for that Space. A process may have rights to access a Space as a *Primitive Space* (in the case that it contains a process stack, code, or local data) or as a *Derived Space* containing persistent objects. Primitive Spaces are protected from invalid read, write, or execute access. A Derived Space can only be accessed by the methods of the persistent objects that it contains.[2] The next section discusses the Choices concept of a process.

4. The Choices Process Concept: Threads

Choices is designed to support real–time multiprocessing and parallel computing on large numbers of processors. The Choices system supports the concept of a computation that is composed of a potentially large number of lightweight parallel processes termed *Threads*. Each Thread represents a small independent sequential computation.

Interrupt and real–time processing requires the ability to switch between Threads with a minimum of context switching overhead. A Thread is implemented with a stack pointer, a program counter and a set of register contents. As the Thread executes, it will access its stack, code, and data from addresses within various Spaces. To accommodate real–time and interrupt processing, Spaces may lock their pages to be resident in physical memory. In addition, a Universe may lock a Space to be resident in virtual memory. A context switch to a Thread that executes and addresses only resident pages in resident virtual memory requires minimal overhead. Interrupt handlers and real–time processes can be implemented in this manner, if desired. Such processes may be protected from other applications by setting the memory protection of the Spaces they access to exclude access in user mode and by running the processes in the supervisor state of the processor. Most Threads, however, will need to access addresses that are not always resident in memory or in the Universe. Switching between Threads of this type will usually involve at least a partial virtual memory context switch.

[2] A Derived Space is created from a Primitive Space by granting processes access rights to the methods of the objects within the Space. In Choices, such objects are called persistent because their existence becomes independent of the lifetime of any one process (see the section on Persistent Objects.) We emphasize the distinction between a Derived Space and a persistent object. Although a Derived Space can contain persistent objects, the Space itself is a Germ object.

A real-time application may use multiple communicating Threads to achieve concurrency and parallelism. A Task is a collection of Threads that have common sets of Spaces to minimize context switching. Little or no virtual memory context switching or memory cache flushing should be needed to switch between two Threads in a Task. Kernel scheduling algorithms can exploit Tasks to achieve high performance. Tasks also provide a framework within which Thread execution may be prioritized; perhaps to optimize the execution of a parallel processing user application.

A *Space Access List* is maintained by the Germ for each Thread. This list specifies the Spaces a Thread must access in order to execute, as well as the access rights that a Thread has to those Spaces. The protection specified by the Space Access List is implemented by a combination of hardware and software. For Primitive Spaces, read, write, and execute protection is provided directly by the Space through the page table and the paging hardware and memory management unit. For Derived Spaces, the protection is achieved by providing "gated" procedure calls to the methods of the objects in the Space. As the Thread invokes a gated call, the call is validated, and if valid, the Space Access List of the Thread is updated to reflect the protection domain of the persistent object. On return, the Space Access List is restored to reflect the original protection domain. The Germ supports efficient operations on the Space Access List that are similar to the rules in a capability model [15, 20].

Communication can be achieved by means of shared Spaces. Popular shared memory and message passing communication schemes exist in the system as part of the operating system provided class hierarchy. Other user defined communication schemes can be built by extending the class hierarchy. An interface compiler for C++ enriches the possible communication schemes. Currently, we have included a Path C++ class (named after Path Pascal [4]), monitors, semaphores, messages, and simple varieties of guarded commands.

Protected communication can be achieved by means of shared Derived Spaces containing persistent objects. The methods of such objects may enforce particular communication protocols upon the Threads that use them and the protection provided the objects prevents misuse.

Since a Thread may execute in any Space, a persistent object may include a Thread and be *active*. Active objects can be used to implement name servers and to send asynchronous messages. Several persistent system objects augment the shared persistent objects and provide high-performance communication channels between Threads and between Threads and devices. System objects are implemented in the Kernel or Germ. They can support stream-based communications, broadcasts, multicasts, and block I/O. Persistent objects are discussed further in the next section.

5. Persistent Objects

Choices is designed with the objective of placing many operating system and subsystem components in a protected Space rather than in a kernel as is done in traditional systems. This is done to reduce the interdependences among operating system components and to increase the coherence of the components themselves. Such components are implemented as Choices *persistent objects*. That is, instances of classes that reside in memory for periods that exceed the execution of a particular Thread and that may be shared between multiple Threads. Persistent objects may be mapped into the virtual memory of several processors at the same time. In a sense, the Germ is a collection of persistent objects that are always resident and accessible in the address space of every processor.

A full description of the protection scheme used in Choices is beyond the scope of this short paper. However, we must introduce enough of the scheme here in order to describe access to and the invocation of a persistent object. Each Thread executes within a protection domain that dictates what the Thread may access. The protection domain of a Thread is dynamic and may change by adding or removing Spaces. Initially, the protection domain depends upon the protection of the executable file that the Thread is created from and the protection domain of the parent Thread. A Thread that executes a method of a persistent object enters a new protection domain that depends upon the protection of the Derived Space and the protection domain of the Thread. When the Thread returns from the method invocation, its previous protection domain is restored.

For example, policy modules of the operating system that traditionally are part of the kernel, may be implemented as persistent objects. A Thread executing one of the methods within these persistent objects may require access to Germ data structures. This is possible by having the Thread enter Supervisor state to execute the method. The gate mechanism changes the protection domain by altering the execution level.

Threads access persistent objects using an *object descriptor* and method. A Thread must obtain the object descriptor before use. Object descriptors are provided from user or system name servers.

Name servers are persistent objects. Choices includes "standard name servers" that are in the Kernel and may be accessed by every Thread. These name servers provide basic facilities like the standard file system and intertask communication. Other user defined name servers must be accessed through the standard name server utilities.

On request, the name server grants the Thread access to the object and returns the object descriptor. The grant operation is implemented in the Germ and checks Kernel protection policy to determine if the name server/Thread grant operation is valid. The name server must have appropriate access rights to the persistent object. If the operation is valid, the Germ adds the Space of the persistent object to the Space Access List of the Thread, updates the Thread's Universe, and returns the Space address and gate information to the name server. The name server packages an object descriptor which includes the persistent object, Space and gate information and returns.

An operation on a persistent object is invoked through a gated request. The Germ ensures that the object descriptor and method used by the Thread gated request correspond to the valid persistent object address and method entry point within the Space. The Space Access List of the Thread is changed to reflect the protection domain requirements of the Space.

In hardware architectures with limited virtual memory, the gated method of invoking a persistent object allows many different Spaces to share the same virtual memory address range. The Space and the persistent objects it contains can be mapped into and out of the same address range on demand[3]. In such implementations, the Space Access List will contain each Space, but only one of the Spaces will be present in the Universe at any one time.

[3] In many hardware architectures, a persistent object must be relocated by a link editor to allow it to execute within a specific address range. This implies that once it is activated, it cannot be moved to a new address range.

6. Exception Handling in Choices

Exceptions in Choices are managed by the *Exception* class and its various subclasses. The parent class of Exception defines the method, *handle*, to manage or correct the exception condition. Upon an exception condition, the Choices Germ manages the task of converting the machine dependent details of exception processing into an invocation of the handle method for the Exception object managing the exception. Various subclasses of Exception define the behavior of *handle* in different ways. Some subclasses of Exception are actually container classes which, based on other inputs, send the handle message to another Exception object contained within.

Two subclasses of Exception of interest are *Trap* and *Interrupt*. The Trap class provides Choices with a mechanism for handling traps that a Thread may generate as a direct result of its execution. This includes machine traps (that is, divide–by–zero or illegal instruction), virtual memory access and protection errors (that is, page faults of various types), and explicit program traps (for example, a "system call").

The basic function of a Trap handler is to, if possible, service the exception condition within the context of the faulting Thread, otherwise to terminate the execution of the faulting Thread.

Interrupts occur asynchronously and, in general, have nothing to do with the currently executing Thread. In Choices, an Interrupt can be awaited by a Thread (and *must* be awaited if it is not to be missed). The *handle* method of the Interrupt class saves the details of the interrupted Thread and resumes the Thread awaiting the occurrence of the interrupt. The Choices Germ has no requirement that all interrupts be handled by the class Interrupt. A Choices kernel implementor can choose to have any type of Exception object handle an interrupt. In future work, various user–oriented exception schemes will be implemented as classes and by the interface compiler. Examples of such schemes can be found in [5].

7. Summary

A Choices Kernel currently runs on a 10 processor Encore Multimax that supports the Store/Space/Universe model of memory management as well as the Task/Thread process concepts. Current effort is devoted towards improvement and further implementation of communication and persistent object support. Future plans include an object–oriented file system, an advanced interface compiler, and tools for configuring Choices systems. Once Choices is stable, the code will be placed in the public domain to promote research into customized operating systems.

References

1. Accetta, Mike, et. al. "Mach: A New Kernel Foundation for UNIX development." USENIX Conference Proceedings, June 1986, pages 93–111.
2. Allchin, J. E. and M. S. McKendry. "Support for Actions and Objects in Clouds: Status Report." Georgia Institute of Technology Technical Report GIT–ICS–83/1, Atlanta, Georgia, January 1983.
3. Brownbridge, D. R., L. F. Marshall, and B. Randell. "The Newcastle Connection, or UNIXes of the World Unite!" Software – Practice and Experience, 1982, pages 1147–1162.
4. Campbell, Roy H. and T. J. Miller. "A Path Pascal Language." Department of Computer Science Technical Report UIUCDCS–R–78–919, University of Illinois at Urbana–Champaign, Urbana, Illinois, April 1978.
5. Campbell R. H. and B. Randell, "Error Recovery in Asynchronous Systems." IEEE Transactions on Software Engineering, Vol. SE–12, No. 8, August, 1986, pp. 811–826.
6. Cheriton, David R. and Willy Zwaenepoel. "Distributed Process Groups in the V Kernel." ACM Transactions on Computer Systems, May 1985, pages 77–107.
7. Encore. "Encore Multimax Technical Summary." Encore Computing Corporation, 1986.
8. Li, Kai and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems." Proceedings of the Fifth Annaul ACM Syposium on Principles of Distributed Computing, August 1986, pages 229–239.
9. Lu, P. M. "A System for Resources Sharing in a Distributed Environment—RIDE." Proceedings of the IEEE Computer Society 3rd COMPSAC, IEEE, New York, 1979.
10. Popek, G., B. Walker, et. al. "LOCUS: A Network Transparent High Reliability Distributed System." Proceedings of the Eighth Symposium on Operating Systems Principles, December, 1981.
11. Rashid, Richard F. "Threads of a New System." UNIX Review, August 1986, pages 37–49.
12. Rifkin, Andrew P., et. al. "RFS Architectural Overview." USENIX Summer Conference Proceedings, Atlanta, Georgia, 1986.
13. Robison, Arch. D. "A Functional Programming Interpreter." M.S. Thesis, Department of Computer Science Technical Report UIUCDCS–R–87–1714, University of Illinois at Urbana–Champaign, Urbana, Illinois, March 1987.
14. Sequent. "Balance 8000 Guide to Parallel Programming." Sequent Computer Systems, Inc., July 1985.
15. Snyder, Lawrence. "Formal Models of Capability–Based Protection Systems." IEEE Transactions on Computers, Vol. C–30, No. 3. March 1981.
16. Swinehart, Daniel, Polle Zellweger, Richard Beach, and Robert Hagmann. "A Structural View of the CEDAR Programming Environment." Transactions on Programming Languages and Systems, October 1986, Vol. 8, No. 4, pages 419–490.
17. Tanenbaum, Andrew S. and Sape J. Mullender. "An Overview of the Amoeba Distributed Operating System." ACM Operating Systems Review, July 1981, pages 51–64.
18. Walsh, Dan, et. al. "Overview of the Sun Network File System." USENIX Conference Proceedings, January 1985, pages 117–124.

19. Wittie, L. D. and A. Van Tilborg. "MICROS – A Distributed Operating System for MICRONET – A Reconfigurable Network Computer" in *Tutorial, Microcomputer Networks*, H. A: Freeman and K. J. Thurber, eds, IEEE Press, 1981, pages 138–147.

20. Wulf, William A., et. al. "HYDRA: The Kernal of a Multiprocessor Operating System." Communications of the ACM, June 1974, pages 337–345.

APPENDIX C

Choices Code

Owner	russo at m.cs.uiuc.edu
Name	../h/Object.h
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:00:52 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 26
Pages printed: 26

Paper size (width, height):
  2560, 3328
Document length:
  60316 bytes
```

```
May 26 05:05 1987    ../h/Object.h Page 1


/*
 * Object.h: definition of the Object parent class (the parent of all classes).
 *
 *      $Header: Object.h,v 11.0 87/05/21 15:49:55 russo Exp $
 *      $Locker:  $
 *
 * The destructor for Object is made virtual so all destructors throughout
 * the system will be likewise. This allows collections of objects
 * to be kept and deleted, while assuring the proper destructors will be
 * called for each class. It increases the size of every object in the
 * system by the size of a pointer  but who cares, memory is cheap.
 */
/*
 * Modification Histroy:
 *      $Log:   Object.h,v $
 * Revision 11.0  87/05/21  15:49:55  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:38  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:05:34  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.4  87/03/29  16:54:12  russo
 * added dummy inline constructor.
 *
 * Revision 8.3  87/03/29  16:47:50  russo
 * added Object as parent class.
 *
 * Revision 8.1  87/03/29  16:35:45  russo
 * initial revision.
 */

#ifndef Object_h
#define Object_h

class Object {

public:
        Object() {};
        virtual ~Object();      // see comment above
};

#endif Object_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 26 05:05 1987    ../h/Assert.h Page 1


/*
 * Assert.h - Assertions.
 *
 *      $Header: Assert.h,v 11.0 87/05/21 15:49:16 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   Assert.h,v $
 * Revision 11.0  87/05/21  15:49:16  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:01  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:05:03  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:28:38  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:45:45  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 1.1  87/02/23  18:31:32  johnston
 * Initial revision
 */

#ifndef Assert_h
#define Assert_h

#ifdef ASSERT

extern void _Assert( char * exp, char * file, int line );
#define Assert(exp)     if( exp ) ; else _Assert( "exp", __FILE__, __LINE__ )

const int NOTREACHED = 0;

#else

#define Assert(exp)

#endif ASSERT

#endif Assert_h
```

```
May 26 05:05 1987  ../h/Debug.h Page 1


/*
 * Debug.h - Debugging stuff
 *
 *      $Header: Debug.h,v 11.0 87/05/21 15:49:33 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   Debug.h,v $
 * Revision 11.0  87/05/21  15:49:33  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:20  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.1  87/04/20  13:42:26  russo
 * changed debug to user CPUPrintf.
 *
 * Revision 9.0  87/04/04  15:05:19  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:29:45  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:07  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 1.1  87/02/23  18:31:32  johnston
 * Initial revision
 */

#ifndef Debug_h
# define Debug_h

extern void Printf( char *, ... );
extern void CPUPrintf( char *, ... );
extern void PanicPrintf( char *, ... );

# ifdef DEBUG

#define Debug   CPUPrintf

# else  DEBUG

// This is int instead of void to avoid "sorry, not implemented" things.
inline /* should be void */ int Debug( char *, ... ) { return (0); }

# endif DEBUG
#endif  Debug_h
```

```
May 26 05:05 1987  ../h/VM.h Page 1


/*
 * VM.h - Virtual memory management (MMU, page tables, virtual addresses, etc.).
 *
 *      $Header: VM.h,v 11.0 87/05/21 15:50:27 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   VM.h,v $
 * Revision 11.0  87/05/21  15:50:27  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:59  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.7  87/04/21  05:42:03  johnston
 * Make shift arguments unsigned to avoid sign extension.
 *
 * Revision 9.4  87/04/20  09:07:26  russo
 * added inlines for conversion from addresses to pages and frames
 *
 * Revision 9.2  87/04/16  16:13:59  johnston
 * Added page/pointer table typedefs and better initialization routine
 * declarations.
 *
 * Revision 9.0  87/04/04  15:06:10  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:30:16  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:47  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 4.0  87/03/10  14:02:37  johnston
 * All new for 1987!
 */

#ifndef VM_h
#define VM_h

#include "md_constants.h"
#include "Debug.h"
#include "Assert.h"
#include "Object.h"             // parent class of VA and PTE (for now)

/*
 * Build inline function returning the value of the named field.
 */
#define FIELDFUNC( name )       unsigned name() \
                                        { return ( data.field.name ); }

/*
 * Virtual address.
 */
```

```
May 26 05:05 1987  ../h/VM.h Page 2


class VA {
        union {
                unsigned all;
                struct {
                        unsigned offset           : 9;  // Page offset.
                        unsigned secondLevelIndex : 7;  // Level 2 index.
                        unsigned firstLevelIndex  : 8;  // Level 1 index.
                        unsigned reserved         : 8;  // RESERVED by hardware.
                } field;
        } data;

        unsigned assign( unsigned all )
        {
                Assert( ((VA *) &all)->data.field.reserved == 0 );
                return ( data.all = all );
        }

        unsigned assign( unsigned l1ix, unsigned l2ix, unsigned offset )
        {
                Assert( l1ix   < 256 );
                Assert( l2ix   < 128 );
                Assert( offset < 512 );
                unsigned va = 0;
                ((VA *) &va)->data.field.firstLevelIndex   = l1ix;
                ((VA *) &va)->data.field.secondLevelIndex  = l2ix;
                ((VA *) &va)->data.field.offset = offset;
                return ( assign( va ) );
        }
public:
        VA()
                { assign( 0 ); }
        VA( unsigned va )
                { assign( va ); }
        VA( VA & va )
                { data.all = va.data.all; }
        VA( unsigned l1ix, unsigned l2ix, unsigned offset )
                { assign( l1ix, l2ix, offset ); }
        VA( char * va )
                { assign( (unsigned) va ); }
        VA( void * va )
                { assign( (unsigned) va ); }

        unsigned operator=( unsigned va )
                { return ( assign( va ) ); }
        unsigned operator=( VA va )
                { return ( data.all = va.data.all ); }
        operator unsigned()
                { return ( data.all ); }

        FIELDFUNC( firstLevelIndex )
        FIELDFUNC( secondLevelIndex )
        FIELDFUNC( offset )

        void printf();
};
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 26 05:05 1987  ../h/VM.h Page 3


/*
 * Address conversions.
 */
inline unsigned int
addrToPage( void * addr )
{
        return( ((unsigned int) addr) >> PAGESHIFT );
}

inline void *
pageToAddr( unsigned int pageNumber )
{
        return( (void *) ( pageNumber << PAGESHIFT ) );
}

inline unsigned int
addrToFrame( void * addr )
{
        return( ((unsigned int) addr) >> 16 );
}

inline void *
frameToAddr( unsigned int frameNumber )
{
        return( (void *) ( frameNumber << 16 ) );
}

/*
 * Page rounding.
 */

overload PageFloor;

inline unsigned int
PageFloor( unsigned n )
{
        return ( n & ~(PAGESIZE - 1) );
}

inline unsigned int
PageFloor( void * a )
{
        return ( PageFloor( (unsigned) a ) );
}

overload PageCeiling;

inline unsigned int
PageCeiling( unsigned n )
{
        unsigned f = PageFloor( n );
        return ( (n == f) ? n : (f + PAGESIZE) );
}

inline unsigned int
PageCeiling( void * a )
```

```
{
        return ( PageCeiling( (unsigned) a ) );
}

/*
 * Page table entry.
 */

const unsigned MAXHANDLERS = 16;        // Only 4 bits to work with in the PTE.

class PTE {
        union {
                unsigned all;
                struct {
                        unsigned valid          :  1;   // Valid.
                        unsigned protectionLevel:  2;   // Protection level.
                        unsigned referenced     :  1;   // Referenced.
                        unsigned modified       :  1;   // Modified.
                        unsigned handlerIndex   :  4;   // Fault handler index.
                        unsigned pageNumber     : 23    // Page frame number.
                } field;
        } data;
public:
        PTE()
                { data.all = 0; }

        PTE( PTE & pte )
        {
                data.all = pte.data.all;
                data.field.referenced = 0;
                data.field.modified = 0;
        }

        PTE( unsigned pte )
        {
                data.all = pte;
                data.field.referenced = 0;
                data.field.modified = 0;
        }

        unsigned operator=( PTE pte )
        {
                data.all = pte.data.all;
                data.field.referenced = 0;
                data.field.modified = 0;
                return ( data.all );
        }

        unsigned operator=( unsigned pte )
        {
                data.all = pte;
                data.field.referenced = 0;
                data.field.modified = 0;
                return ( data.all );
        }
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        operator unsigned()
                { return ( data.all ); }

        void map( unsigned pn, unsigned pl )
        {
                Assert( pn < 0x8000 );          // Page number in range.
                Assert( pl < 0x4 );             // Protection level in range.
                Assert( !data.field.valid );    // Page not already mapped.
                data.field.pageNumber = pn;
                data.field.protectionLevel = pl;
                data.field.valid = 1;
                data.field.referenced = 0;
                data.field.modified = 0;
        }

        void map( unsigned pn )
        {
                Assert( pn < 0x8000 );          // Page number in range.
                Assert( !data.field.valid );    // Page not already mapped.
                data.field.pageNumber = pn;
                data.field.valid = 1;
                data.field.referenced = 0;
                data.field.modified = 0;
        }

        void unmap()
        {
                Assert( data.field.valid );
                data.field.valid = 0;
        }

        void handle( unsigned hi, unsigned pl )
        {
                Assert( hi < MAXHANDLERS );     // Handler index in range.
                Assert( pl < 0x4 );             // Protection level in range.
                data.field.handlerIndex = hi;
                data.field.protectionLevel = pl;
                data.field.referenced = 0;
                data.field.modified = 0;
        }

        FIELDFUNC( pageNumber );
        FIELDFUNC( handlerIndex );
        FIELDFUNC( modified );
        FIELDFUNC( referenced );
        FIELDFUNC( protectionLevel );
        FIELDFUNC( valid );

        void printf();
};

/*
 * Page table initialization routines.
 */

typedef PTE PageTable[256];
```

```
May 26 05:05 1987  ../h/VM.h Page 6


typedef PTE PointerTable[128];

extern void InitPageTable( PageTable & );
extern void InitPointerTable( PointerTable & );

extern PTE * InitFirstLevelPageTable( PTE * );          // Walking history...
extern PTE * InitSecondLevelPageTable( PTE * );         // Ditto.

/*
 * MMU registers.
 */

/*
 * Error/Invalidate Address.
 */

class EIA {
        union {
                unsigned all;
                struct {
                        unsigned address        : 24;   // Fault address.
                        unsigned reserved       :  7;   // RESERVED by hardware.
                        unsigned txPTB          :  1;   // PTB0/1 did translate.
                } field;
        } data;

        unsigned assign( unsigned eia )
        {
                return ( data.all = eia );
        }

        unsigned assign( unsigned txPTB, unsigned address )
        {
                Assert( (txPTB == 0) || (txPTB == 1) );
                Assert( address < 0x1000000 );
                unsigned eia = 0;
                ((EIA *) &eia)->data.field.txPTB = txPTB;
                ((EIA *) &eia)->data.field.address = address;
                return( assign( eia ) );
        }
public:
        EIA()
                { assign( 0 ); }
        EIA( unsigned eia )
                { assign( eia ); }
        EIA( EIA & eia )
                { data.all = eia.data.all; }
        EIA( unsigned txPTB, unsigned address )
                { assign( txPTB, address ); }

        unsigned operator=( unsigned eia )
                { return( assign( eia ) ); }
        unsigned operator=( EIA eia )
                { return( data.all = eia.data.all ); }
        operator unsigned()
                { return ( data.all ); }
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 26 05:05 1987  ../h/VM.h Page 7


        unsigned address()
                { return ( data.field.address ); }
        unsigned txPTB()
                { return ( data.field.txPTB ); }

        void printf();
};

/*
 * Memory status register.
 */

class MSR {
        union {
                unsigned all;
                struct {
                   unsigned txError        : 1;  // Address translation error.
                   unsigned magic          : 1;  // Clear MSR.
                   unsigned BPTError       : 1;  // Breakpoint error.
                   unsigned ProtLevelError: 1;  // Protection level error.
                   unsigned L1PTEError     : 1;  // First level PTE error.
                   unsigned L2PTEError     : 1;  // Second level PTE error.
                   unsigned BPR            : 1;  // BPR 0/1 caused error.
                   unsigned reserved1      : 1;  // RESERVED by hardware.
                   unsigned readError      : 1;  // Write/read error
                   unsigned BPTReadError   : 1;  // Write/read breakpoint error.
                   unsigned txStatError    : 3;  // Translation bus cycle error?
                   unsigned BPTStatError   : 3;  // Breakpoint bus cycle error?
                   unsigned txUser         : 1;  // Translate user addresses.
                   unsigned txSupervisor   : 1;  // Translate supervisor addrs.
                   unsigned userPTB        : 1;  // Use PTB0/1 for user.
                   unsigned override       : 1;  // Use super. prots. for user.
                   unsigned BPTEnable      : 1;  // Enable breakpoints.
                   unsigned BPTUserOnly    : 1;  // Breakpoint in user mode only.
                   unsigned ai             : 1;  // Abort/NMI trap?
                   unsigned flowTrace      : 1;  // Enable flow tracing.
                   unsigned flowUserOnly   : 1;  // Flow trace in user mode only.
                   unsigned nonseqTrap     : 1;  // Enable nonseq. flow traps?
                   unsigned reserved2      : 5;  // RESERVED by hardware.
                } field;
        } data;

        unsigned assign( unsigned msr )
        {
                Assert( ((MSR *) &msr)->data.field.reserved1 == 0  );
                Assert( ((MSR *) &msr)->data.field.reserved2 == 0  );
                return ( data.all = msr );
        }

        unsigned assign( unsigned magic, unsigned txUser
                         unsigned txSupervisor  unsigned userPTB )
                {
                        Assert( (magic == 0  || (magic == 1) );
                        Assert( (txUser == 0) || (txUser == 1) );
                        Assert( (txSupervisor == 0) ||  txSupervisor == 1) );
```

```
                                Assert( userPTB == 0 || userPTB == 1 );
                                unsigned msr = 0;
                                ((MSR *) &msr)->data.field.magic = magic;
                                ((MSR *) &msr)->data.field.txUser = txUser;
                                ((MSR *) &msr)->data.field.txSupervisor = txSupervisor;
                                ((MSR *) &msr)->data.field.userPTB = userPTB;
                                return( assign( msr ) );
                        }
public:
        MSR()
                { assign( 0 ); }
        MSR( unsigned msr )
                { assign( msr ); }
        MSR( MSR & msr )
                { data.all = msr.data.all; }
        MSR( unsigned magic, unsigned txUser,
             unsigned txSupervisor, unsigned userPTB )
                { assign( magic, txUser, txSupervisor, userPTB ); }

        unsigned operator= ( unsigned msr )
                { return( assign( msr ) ); }
        unsigned operator= ( MSR msr )
                { return( data.all = msr.data.all ); }
        operator unsigned()
                { return( data.all ); }

        FIELDFUNC( txError );
        FIELDFUNC( magic );
        FIELDFUNC( BPTError );
        FIELDFUNC( ProtLevelError );
        FIELDFUNC( L1PTEError );
        FIELDFUNC( L2PTEError );
        FIELDFUNC( BPR );
        FIELDFUNC( reserved1 );
        FIELDFUNC( readError );
        FIELDFUNC( BPTReadError );
        FIELDFUNC( txStatError );
        FIELDFUNC( BPTStatError );
        FIELDFUNC( txUser );
        FIELDFUNC( txSupervisor );
        FIELDFUNC( userPTB );
        FIELDFUNC( override );
        FIELDFUNC( BPTEnable );
        FIELDFUNC( BPTUserOnly );
        FIELDFUNC( ai );
        FIELDFUNC( flowTrace );
        FIELDFUNC( flowUserOnly );
        FIELDFUNC( nonseqTrap );
        FIELDFUNC( reserved2 );

        void printf();
};

/*
 * MMU register read/write routines.
 */
```

```
extern void WriteEIA( EIA eia );        // THESE 4 ARE SUSPICIOUS
extern unsigned ReadEIA();
extern void WriteMSR( MSR msr );
extern unsigned ReadMSR();

extern void WritePTB0( void * ptb );
extern void WritePTB1( void * ptb );
extern void * ReadPTB0();
extern void * ReadPTB1();

#endif VM_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * Store.h - Physical memory allocation.
 *
 *      $Header: Store.h,v 11.0 87/05/21 15:50:13 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   Store.h,v $
 * Revision 11.0  87/05/21  15:50:13  russo
 * Console input and private stores.
 *
 * Revision 10.4  87/05/15  06:53:34  johnston
 * Added another inStore.
 *
 * Revision 10.2  87/05/15  06:17:59  johnston
 * Fixed offset problem in the mark functions.
 *
 * Revision 10.1  87/05/15  04:43:46  johnston
 * Redid Store.
 *
 * Revision 10.0  87/04/22  07:33:45  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.1  87/04/15  15:39:34  johnston
 * Fixed constructor to use physical memory pages to allocate itself.
 *
 * Revision 9.0  87/04/04  15:06:00  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:30:04  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:30  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 1.1  87/02/23  18:31:34  johnston
 * Initial revision
 */

#ifndef Store_h
#define Store_h

#include "Assert.h"
#include "Debug.h"
#include "Lock.h"
#include "Object.h"
#include "VM.h"

class Store : public Object {
        Lock lock;
        unsigned basePage;
        unsigned highPage;
        unsigned freePageCount;
        unsigned setEntryCount;
        unsigned pagesPerSetEntry;
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        unsigned set[1];

        void mark( unsigned page );
        void unmark( unsigned page );
        int marked( unsigned page );

        unsigned nextFree( unsigned lowPage );
        unsigned contiguous( unsigned lowPage, unsigned pageCount );
public:
        Store( unsigned basePage,
               unsigned pageCount,
               unsigned * stateBasePage );
        ~Store();

        char * allocate( unsigned pageCount );
        void deallocate( char * baseAddr, unsigned pageCount );
        void reserve( unsigned basePage, unsigned pageCount );

        int inStore( unsigned page );
        int inStore( char * addr );
};

inline int
Store::inStore( unsigned page )
{
        return ( ( page >= this->basePage ) && ( page < this->highPage ) );
}

inline int
Store::inStore( char * addr )
{
        return ( this->inStore( addrToPage( (char *) PageFloor( addr ) ) ) );
}

inline int
Store::marked( unsigned page )
{
        Assert( this->inStore( page ) );
        unsigned offsetPage = page - this->basePage;
        return ( this->set[ offsetPage / this->pagesPerSetEntry ] &
                ( 0x1 << ( offsetPage % this->pagesPerSetEntry ) ) );
}

inline void
Store::mark( unsigned page )
{
        Assert( this->inStore( page ) );
        Assert( ! this->marked( page ) );
        unsigned offsetPage = page - this->basePage;
        this->set[ offsetPage / this->pagesPerSetEntry ] |=
                ( 0x1 << ( offsetPage % this->pagesPerSetEntry ) );
        Assert( this->marked( page ) );
        Assert( this->freePageCount != 0 );
        this->freePageCount--;
}
```

```
May 26 05:05 1987   ../h/Store.h Page 3


inline void
Store::unmark( unsigned page )
{
        Assert( this->inStore( page ) );
        Assert( this->marked( page ) );
        unsigned offsetPage = page - this->basePage;
        this->set[ offsetPage / this->pagesPerSetEntry ] &=
                ~( 0x1 << ( offsetPage % this->pagesPerSetEntry ) );
        Assert( ! this->marked( page ) );
        Assert( this->freePageCount < ( this->highPage - this->basePage ) );
        this->freePageCount++;
}

#endif
```

```
May 26 05:05 1987  ../h/md_tuneable.h Page 1


/*
 * md_tuneable.h - Machine-dependent, tuneable parameters.
 *
 *      $Header: md_tuneable.h,v 11.0 87/05/21 15:50:40 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   md_tuneable.h,v $
 * Revision 11.0  87/05/21  15:50:40  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:34:08  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:06:19  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:30:25  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:58  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 4.1  87/03/09  12:42:47  johnston
 * Initial revision.
 */

#ifndef md_tuneable_h
#define md_tuneable_h

/*
 * Page frame ranges and addresses.
 *
 * NOTE: A "frame" is the space mapped by a single *first-level* page table
 *      entry (64k bytes, here).
 */
const int NFRAMES = 256;                // Complete address space (0..255).
                                        // This is 64k * 256 = 16M bytes.
                                        // This *must* be <= 256.

const int GKLOWFRAME =          0x000000;               // Germ/Kernel (0..126).
const int GKLOWPAGE  =          0x000000;
const int GKLOWADDR  =          0x000000;

const int GKHIGHFRAME =         0x00007f;
const int GKHIGHPAGE =          GKHIGHFRAME <<  7;
const int GKHIGHADDR =          GKHIGHFRAME << 16;

const int STACKLOWFRAME =       GKHIGHFRAME;            // System stack (127).
const int STACKLOWPAGE   =      GKHIGHPAGE;
const int STACKLOWADDR =        GKHIGHADDR;

const int STACKHIGHFRAME =      0x000080;
const int STACKHIGHPAGE  =      STACKHIGHFRAME <<  7;
const int STACKHIGHADDR =       STACKHIGHFRAME << 16;
```

```
May 26 05:05 1987  ../h/md_tuneable.h Page 2


const int TASKLOWFRAME =         STACKHIGHFRAME;          // Task (128..251).
const int TASKLOWPAGE =          STACKHIGHPAGE;
const int TASKLOWADDR =          STACKHIGHADDR;

const int TASKHIGHFRAME =        0x0000fc;
const int TASKHIGHPAGE =         TASKHIGHFRAME <<  7;
const int TASKHIGHADDR =         TASKHIGHFRAME << 16;

const int THREADSTACKFRAME = 251;        // Thread stack (inside of Task space)

const int HWLOWFRAME =           TASKHIGHFRAME;           // HW (252..255).
const int HWLOWPAGE =            TASKHIGHPAGE;
const int HWLOWADDR =            TASKHIGHADDR;

const int HWHIGHFRAME =          0x000100;
const int HWHIGHPAGE =           HWHIGHFRAME <<  7;
const int HWHIGHADDR =           HWHIGHFRAME << 16;

#endif md_tuneable_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 26 05:05 1987  ../h/mi_tuneable.h Page 1


/*
 * mi_tuneable.h: Machine-independent, tuneable parameters.
 *
 *      $Header: mi_tuneable.h,v 11.0 87/05/21 15:50:43 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   mi_tuneable.h,v $
 * Revision 11.0  87/05/21  15:50:43  russo
 * Console input and private stores.
 *
 * Revision 10.1  87/05/07  05:42:15  johnston
 * Changed MAXCPUS from 32 to 64.
 *
 * Revision 10.0  87/04/22  07:34:11  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.1  87/04/15  15:45:40  johnston
 * Added MAXKERNELS.
 *
 * Revision 9.0  87/04/04  15:06:21  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:30:27  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:47:01  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 4.1  87/03/09  12:43:02  johnston
 * Initial revision.
 */

#ifndef mi_tuneable_h
#define mi_tuneable_h

#define MAXCPUS         64
#define MAXKERNELS       2

#endif mi_tuneable_h
```

May 26 05:05 1987 ../h/md_constants.h Page 1

```
/*
 * md_constants.h - Machine-dependent constants.
 *
 *      $Header: md_constants.h v 11.0 87/05/21 15:50:37 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   md_constants.h,v $
 * Revision 11.0  87/05/21  15:50:37  russo
 * Console input and private stores.
 *
 * Revision 10.1  87/05/16  23:35:35  russo
 * added LASTADDRESSABLELOCATION
 *
 * Revision 10.0  87/04/22  07:34:05  russo
 * New Spaces. Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:06:16  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:30:21  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:54  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts
 *
 * Revision 4.1  87/03/09  12:42:25  johnston
 * Initial revision.
 */

#ifndef md_constants_h
#define md_constants_h

#define INITIALMSR      0x00070002      /* Initial value to load into the MSR */

#define LASTADDRESSABLELOCATION 0x00ffffff

#define PAGESIZE        512             /* Number of bytes per page */
#define PAGESHIFT       9               /* Number of bits for page offset */

#define PSR_U_BIT 8
#define PSR_U 256
#define PSR_C_BIT 0
#define PSR_C 1
#define PSR_I_BIT 11
#define PSR_I 2048
#define PSR_S_BIT 9
#define PSR_S 512
#define PSR_S_BIT 9
#define PSR_S 512
#define PSR_T_BIT 1
#define PSR_T 2
#define PSR_F_BIT 5
#define PSR_F 32
```

May 26 05:05 1987 ../h/md_constants.h Page 2

```
/*
 * DPC control/status registers.
 */
#define DPC_STATUS              0xfffffffc      /* DPC status register. */
#define DPC_CONTROL             0xfffffe48      /* DPC control register. */

#define DPC_STATUS_VBUS_BUSY    0x8000000       /* Vector bus busy. */

/*
 * DPC Vector Bus registers.
 */
#define DPC_VBUS_CLASS          0xfffffe22      /* Class register. */
#define DPC_VBUS_TX             0xfffffe24      /* Transmit register. */

/*
 * Encore stuff -+
 *              |
 *              |
 *              V
 */

#define IO_BASE         0x800000        /* Base of I/O space */
#define I534_BASE       0x800060        /* Base if 4-line asynch card */
#define SL_BASE         I534_BASE       /* Serial line base */
#define SYS_CLOCK       (I534_BASE+4)   /* System clock address */
#define COUNTER         (I534_BASE+6)   /* Free running counter address */
#define MAXCOUNTER      0xff            /* Maximum value of counter */

#define DSD_WADDR       0x00bc00        /* DSD wakeup address */
#define DSD_WIO         0x800bc0        /* DSD wakeup I/O address */

#define ICU_BASE        0xfffe00        /* Interrupt Control Unit */
#define IL_BASE         0x800000        /* InterLAN NI3010 */

#define PPI_BASE        0xc00020        /* Base of Parallel Ports */
#define CPU_NMIREG      PPI_BASE        /* CPU nmi register */
#define GENNMI          0x0d            /* Generate nmi code */
#define ARMNMI          0x0c            /* Arm nmi code */
#define CPU_IPIREG      (PPI_BASE+2)    /* Intercpu interrupt register */
#define CPU_IDREG       (PPI_BASE+4)    /* CPU identifier register */
#define CPU_ID          0x7             /* Mask for cpu id */
#define PPI_CTL         (PPI_BASE+6)    /* Control register */
#define PPINIT          0x81            /* Init code */
#define ROMKILL         0x0e            /* Kill rom code code */

#define CPU_SWREG       0xc00030        /* CPU switch register */
#define SW_BOOTCPU      7               /* Boot processor */
#define SW_CLOCK        6               /* Clock interrupt processor */
#define SW_NET          5               /* Network interrupt cpu */
#define SW_DISK         4               /* Disk interrupt cpu */
#define SW_SERIAL       3               /* Serial line interrupt cpu */
#define CPU_MCHECKREG   0xc00030        /* Machine check register */

#define CPU_MPARREG_BASE 0x800040       /* Memory parity register base */

#define NUMSYSINTRS 256
```

May 26 05:05 1987 ../h/md_constants.h Page 3

```
#define NUMSYSTRAPS 16
#define SCCMEM_BASE_24 16515072
#define _FBcounter 0-196096
#define DPCREG_CTL -440
#define DPCREG_NMI -504
#define DPCREG_STS -4
#define DPCREG_BASE -512
#define DPCREG_SENDVEC -476
#define DPCCTL_NMI_DISABLE 4096
#define DPCCTL_GEN_SYSNMI 2097152
#define DPCCTL_FIX 60830208
#define DPCSTS_CPUID 1
#define DPCSTS_SLOTID 60
#define DPCSTS_OUTPUT_READY 268435456
#define DPCSTS_VECBUS_TXREQ_BIT 27
#define DPCSTS_FIX -536858112
#define DPCREG_VECTOR -512
#define DPCREG_FIFO -472
#define DPCSTS_ISBX_PRESENT 1073741824
#define DPCREG_SBXCTL0 -319
#define DPCREG_SBXDATA0 -315
#define DPCREG_SBXCTL1 -311
#define DPCREG_SBXDATA1 -307
#define DPCREG_TSEVECWRITE -465
#define DPCREG_TSECTL -451
#define DPCREG_TSECNT1 -459
#define DPCREG_TSECNT2 -455
#define DPC_FIFOSIZE 17
#define DPCSBXCTL_TXRDY_BIT 2
#define DPCSBXCTL_RXRDY_BIT 0

#endif md_constants_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

May 26 05:05 1987 ../h/Task.h Page 1

```
/*
 * Task.h: task class description
 *
 *      $Header: Task.h,v 11.0 87/05/21 15:50:16 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Task.h,v $
 * Revision 11.0  87/05/21  15:50:16  russo
 * Console input and private stores.
 *
 * Revision 10.3  87/05/12  15:06:54  russo
 * added initialThread member function
 *
 * Revision 10.0  87/04/22  07:33:48  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.3  87/04/20  13:25:57  russo
 * added a lock to the instances
 *
 * Revision 9.0  87/04/04  15:06:02  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:30:08  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:36  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 4.1  87/03/09  16:13:10  russo
 * initial revision.
 */

#ifndef Task_h
#define Task_h

#include "Object.h"     // parent class
#include "FaultHandler.h"
#include "Lock.h"
#include "Space.h"
#include "Thread.h"

typedef void (* TPFV)();

class Task : public Object {
        Space * space;
        Lock lock;
        FaultHandler * stackFaultHandler;
        Thread * threads;
public:
        Task( Space * space, TPFV initialEntryPoint );
        ~Task();
        Thread * initialThread();
        Thread * startThread( TPFV entryPoint, int argument );
        Space * getSpace() { return( space ); }
```

```
#endif Task_h
```

```
/*
 * Thread.h: thread class description
 *
 *      $Header: Thread.h,v 11.2 87/05/21 23:25:39 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   Thread.h,v $
 * Revision 11.2  87/05/21  23:25:39  russo
 * removed unneded method
 *
 * Revision 11.1  87/05/21  23:23:10  russo
 * added instance variable to save the initial USP.
 *
 * Revision 11.0  87/05/21  15:50:19  russo
 * Console input and private stores.
 *
 * Revision 10.13  87/05/13  18:55:39  russo
 * made GermThread a subclass of Thread, not KernelThread.
 *
 * Revision 10.11  87/05/12  09:50:59  russo
 * added new GermThread subclass
 *
 * Revision 10.10  87/05/10  21:05:28  russo
 * each thread now carries around a small interrupt stack of its own.
 *
 * Revision 10.0  87/04/22  07:33:51  russo
 * New Spaces, Universes and CPU objects work. Finally!
 *
 * Revision 9.0  87/04/04  15:06:05  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.9  87/04/01  16:15:00  russo
 * added offsets for assembler to use
 *
 * Revision 8.0  87/03/29  15:30:11  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:41  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 1.1  87/02/23  18:31:39  johnston
 * Initial revision
 */

#ifndef Thread_h
#define Thread_h

#include "Object.h"     // parent class
#include "Space.h"

const int stackSize = (512 - 32);

class Thread : public Object {
protected:
```

```
May 26 05:05 1987  ../h/Thread.h Page 2

        char    interruptStack[stackSize];

        char *  stackPointer;
        char *  initialUSP;
        int priority;
        void * kernelInfo;
        typedef void (* PFV)();
        Space * lspaces;        // list of spaces this Thread needs to run
public:
        Thread * next;  // for linking the Thread into lists
        Thread * last;  // for linking the Thread into lists

        Thread ( PFV, int *, int, int, void * );
        ~Thread ();

        char * interruptStackPointer() { return( stackPointer ); }
        void setInterruptStackPointer( char * isp )
                { stackPointer = isp; }
        Space * spaces() { return( lspaces ); }
        void setSpaces( Space * s ) { lspaces = s; }
        void * getKernelInfo() { return( kernelInfo ); }
        int getPriority() { return( priority ); }
        void dump();
        virtual int isPreemptable();
};

class KernelThread : public Thread {
protected:
public:
        KernelThread( PFV, int *, int, int, void * );
        ~KernelThread();
};

class GermThread : public Thread {
protected:
public:
        GermThread( GermThread *, PFV, int *, int, int, void * );
        ~GermThread();
};

class InterruptThread : public KernelThread {
protected:
public:
        InterruptThread( PFV, int *, int, int, void * );
        ~InterruptThread();
        int isPreemptable();
};

#endif Thread_h
```

```
May 26 05:05 1987  ../h/CPU.h Page 1


/*
 * CPU.h: per-cpu private information class
 *
 *      $Header: CPU.h,v 11.1 87/05/21 16:41:50 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   CPU.h,v $
 * Revision 11.1  87/05/21  16:41:50  russo
 * only need a single thread to delete not a queue.
 *
 *
 * Revision 11.0  87/05/21  15:49:29  russo
 * Console input and private stores.
 *
 * Revision 10.30  87/05/17  13:57:09  russo
 * added DeleteQueue member
 *
 * Revision 10.27  87/05/11  17:39:37  russo
 * added local and global store fields.
 *
 * Revision 10.22  87/05/06  16:52:47  russo
 * made number of vectored exceptions a member function
 *
 * Revision 10.21  87/05/04  19:09:27  russo
 * added id() method
 *
 * Revision 10.18  87/04/27  19:14:11  russo
 * added Number of vectored exceptions const
 *
 * Revision 10.10  87/04/26  21:22:29  russo
 * added Exception stuff
 *
 * Revision 10.0  87/04/22  07:33:17  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.9  87/04/21  15:11:57  russo
 * added interrupt stack member.
 * really VirtualSetUp should be a friend to save a lot of set function
 * that should never be called by anyone else.
 *
 * Revision 9.1  87/04/21  09:53:21  russo
 * initial revision
 */

#ifndef CPU_h
#define CPU_h

#include "mi_tuneable.h"
#include "Space.h"
#include "Thread.h"
#include "Universe.h"
#include "Exception.h"
#include "Scheduler.h"
```

```
                // Really 256 - 16 but keep it small for now
const int nExceptions = 20;

class CPU : public Object {

protected:
        Universe        * cpuUniverse;
        Thread          * cpuCurrentThread;
        Thread          * cpuIdleThread;        // I'm not sure about this
        Space           * cpuHeapSpace;
        Exception       * cpuThings[nExceptions];
        Store           * cpuPrivateStore;
        Store           * cpuGlobalStore;
        Scheduler       * cpuScheduler;
        Thread          * cpuThreadToDelete;

public:
        CPU( CPU * where );
        ~CPU();

        /*
         * access functions.
         */
        int numberOfVectoredExceptions()
                { return( nExceptions ); }
        Universe * universe()
                { return( this->cpuUniverse ); }
        Thread * currentThread()
                { return( this->cpuCurrentThread ); }
        Thread * idleThread()
                { return( this->cpuIdleThread ); }
        Space * heapSpace()
                { return( this->cpuHeapSpace ); }
        Exception * exception( int vector
                { return( this->cpuThings[vector] ); }
        Store * privateStore()
                { return( this->cpuPrivateStore ); }
        Store * globalStore( )
                { return( this->cpuGlobalStore ); }
        Scheduler * scheduler()
                { return( this->cpuScheduler ); }
        Thread * threadToDelete()
                { return( this->cpuThreadToDelete ); }

        /*
         * set-value functions.
         */
        void setUniverse( Universe * U ) {
                this->cpuUniverse = U;
        }
        void setCurrentThread( Thread * aThread ) {
                this->cpuCurrentThread = aThread;
        }
        void setIdleThread( Thread * aThread ) {
                this->cpuIdleThread = aThread;
        }
```

```
        void setHeapSpace( Space * space   {
                this->cpuHeapSpace = space;
        }
        void setException( int vector, Exception * e ) {
                cpuThings[vector] = e;
        }
        void setPrivateStore( Store * store ) {
                this->cpuPrivateStore = store;
        }
        void setGlobalStore( Store * store ) {
                this->cpuGlobalStore = store;
        }
        void setScheduler( Scheduler * scheduler ) {
                this->cpuScheduler = scheduler;
        }
        void setThreadToDelete( Thread * thread ) {
                this->cpuThreadToDelete = thread;
        }

        /*
         * Others.
         */
        unsigned int id();
};

extern CPU * Me;

#endif CPU_h
```

```
May 26 05:05 1987   ../h/SVCs.h Page 1


/*
 * SVCs.h: definition of SVC numbers the kernel recognizes.
 *
 * This file should go away when we add cross domain object calls
 *
 *      $Header: SVCs.h,v 11.1 87/05/24 22:28:09 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   SVCs.h,v $
 * Revision 11.1  87/05/24  22:28:09  russo
 * changed order of defines?
 *
 * Revision 11.0  87/05/21  15:50:02  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:42  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:05:53  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:29:58  russo
 * _new and _delete added for memory management. Also  class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:16  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 1.1  87/03/17  14:07:50  russo
 * Initial revision
 */

#ifndef SVCs_h
#define SVCs_h

#define PRINTF_SVC       0
#define KILLTHREAD_SVC   1
#define KILLTASK_SVC     2
#define STARTTHREAD_SVC  3

#define LAST_SVC         3

#endif SVCs_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 26 05:05 1987  ../h/FaultHandler.h Page 1


/*
 * FaultHandler.h - Fault handler class definition.
 *
 *      $Header: FaultHandler.h,v 11.1 87/05/24 05:07:57 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   FaultHandler.h,v $
 * Revision 11.1  87/05/24  05:07:57  russo
 * adjusting tto reflect new ideas about fault handlers.
 *
 * Revision 11.0  87/05/21  15:49:38  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:23  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:05:21  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:29:49  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:46:09  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 6.1  87/03/22  12:21:48  russo
 * initial revision
 */

#ifndef FaultHandler_h
#define FaultHandler_h

#include "Object.h"     // parent class
#include "Store.h"

class Space;    // include Space.h would cause a circular definition.

/*
 * Fault Handler Parent Class. This just defines what the rest of the kernel
 * thinks a fault handler interface looks like. Individual derived types
 * can specify all kinds of way to handle the actual faults, as long as
 * they meet the interface described here. The parent class implementation
 * of fixFault currently Halts the processor.
 */
class FaultHandler : public Object {
public:
        virtual void fixFault( Space * space, void * address );
};

/*
 * Fault handler to manage allocation/deallocation from a store.
 * This is about as simple as they get.
 */
class StoreManager : public FaultHandler {
```

```
protected:
        Store * storeBeingManaged;
public:
        StoreManager( Store * );
        ~StoreManager();
        void fixFault( Space * space, void * address );
};

/*
 * Fault handler subclass to fill on demand a faulting page.
 * The only thing actually implemented here is the code to allocate a free
 * page from the store and then map it into the current tasks address
 * space at the faulting address.
 */
class DemandFillFaultHandler : public FaultHandler {
public:
        void allocateAndMap( Space * space, void * address );
};

/*
 * A subclass of demand fill fault handler that takes a filler in the
 * constructor that defines how to fill the faulting page once it is
 * allocated and mapped in.
 */
#include "Filler.h"

class DemandFillerClassFaultHandler : public DemandFillFaultHandler {
        Filler * filler;
public:
        DemandFillerClassFaultHandler( Filler * filler );
        ~DemandFillerClassFaultHandler();
        void fixFault( Space * space, void * address );
};

/*
 * A subclass of DemandFillFaultHandler that fills a faulting page with
 * zeros once it is allocated and mapped in.
 */
class DemandZeroFaultHandler : public DemandFillFaultHandler {
public:
        DemandZeroFaultHandler();
        ~DemandZeroFaultHandler();
        void fixFault( Space * space, void * address );
};

#endif FaultHandler_h
```

```
/*
 * File.h: the parent file class definition.
 *
 *      $Header: File.h,v 11.0 87/05/21 15:49:44 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   File.h,v $
 * Revision 11.0  87/05/21  15:49:44  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:26  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:05:25  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:29:52  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  15:15:11  russo
 * brought revision number up to date
 *
 * Revision 1.1  87/03/25  15:15:01  russo
 * Initial revision
 */

#ifndef File_h
#define File_h

#include "Object.h"     // parent class

extern void Halt();

class File : public Object {
public:
        virtual int readRecords( long startRecord, void * buffer, int count );
        virtual int writeRecords( long startRecord, void * buffer, int count );
        virtual int getRecordSize();
};

class MemoryFile : public File {
        char * location;
        int length;
public:
        MemoryFile( char * location, int length );
        ~MemoryFile();

        int readRecords( long startRecord, void * buffer, int count );
        int writeRecords( long startRecord, void * buffer, int count );
};

#endif File_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * Filler.h - Filler class definition.
 *
 *      $Header: Filler.h,v 11.0 87/05/21 15:49:47 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   Filler.h,v $
 * Revision 11.0  87/05/21  15:49:47  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:28  russo
 * New Spaces, Universes and CPU objects work. Finally!
 *
 * Revision 9.0  87/04/04  15:05:27  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:29:54  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.1  87/03/28  15:44:35  russo
 * Filler class definition move here from FaultHandler.h
 */

#ifndef Filler_h
#define Filler_h

#include "Object.h"     // parent class

/*
 * Filler parent class for use by the DemandFillerClassFaultHandler class.
 * Maybe the default fillPage() can do something nice like fill it with
 * zeros? Halting seems a little drastic.
 */
extern void Halt();

class Filler : public Object {
public:
        virtual void fillPage( void * faultingAddress ) {
                Printf( "Filler::fillPage(%x) called!!\n",
                        faultingAddress );
                Halt();
        }
};

/*
 * Filler to fill a COFF section from a COFF file.
 */
#include "File.h"

class COFFSectionFiller : public Filler {
        File * file;
        void * sectionStart;
        int sectionLength;      // in bytes.
        long fileLocation;      // the location into the COFF file to load from.
```

```
public:
        COFFSectionFiller( File * file, void * start, int length,
                           long location );
        ~COFFSectionFiller();
        void fillPage( void * address );
};

#endif Filler_h
```

```
/*
 * Lock.h - low-level spin lock.
 *
 *      $Header: Lock.h,v 11.7 87/05/26 05:53:59 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   Lock.h,v $
 * Revision 11.7  87/05/26  05:53:59  russo
 * Made heldByAnother an outline.
 *
 * Revision 11.6  87/05/25  19:31:57  johnston
 * Added heldByAnother().
 *
 * Revision 11.5  87/05/25  18:38:09  johnston
 * Added heldBy().
 *
 * Revision 11.4  87/05/25  06:52:17  johnston
 * Made heldByMe a non-inline.
 *
 * Revision 11.3  87/05/25  06:22:59  johnston
 * lksdjflskjdflj
 *
 * Revision 11.2  87/05/25  06:17:13  johnston
 * Fixed declaration of ThisCPU().
 *
 * Revision 11.1  87/05/25  05:57:58  johnston
 * Added heldByMe().
 *
 * Revision 11.0  87/05/21  15:49:53  russo
 * Console input and private stores.
 *
 * Revision 10.3  87/05/13  20:57:18  johnston
 * Removed Lock::free() because it's confusing.
 *
 * Revision 10.0  87/04/22  07:33.30  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:05:29  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.7  87/04/03  06:35:32  russo
 * CFRONT is screwed up
 *
 * Revision 8.5  87/04/02  22:02:35  johnston
 * Added inline definitions of held() and free().
 *
 * Revision 1.1  87/04/02  16:19:59  johnston
 * Initial revision
 */

#ifndef Lock_h
#define Lock_h

extern unsigned ThisCPU();
```

```
class Lock {
        unsigned char state;            // The actual lock.
        unsigned interruptState;        // Interrupts on flag.
        unsigned holdingCPU;            // CPUID holding lock (if locked).
public:
        Lock();

        void acquire();
        void release();

        /*
         * The use of any of these will probably causes races...beware.
         */
        int held()
                { return ( this->state & 0x1 ); }
        unsigned heldBy()
                { return ( this->holdingCPU ); }
        int heldByMe()
                { return ( this->holdingCPU == ThisCPU() ); }
        int heldByAnother();
};

#endif Lock_h
```

```
May 26 05:05 1987  ../h/Timer.h Page 1


/*
 * Timer.h - Per-CPU timer.
 *
 *      $Header: Timer.h,v 11.0 87/05/21 15:50:22 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   Timer.h,v $
 * Revision 11.0  87/05/21  15:50:22  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:33:55  russo
 * New Spaces. Universes and CPU objects work. Finally!
 *
 * Revision 9.0  87/04/04  15:06:08  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 1.1  87/04/03  09:19:25  johnston
 * Initial revision
 */

#ifndef Timer_h
#define Timer_h

#include "Object.h"

class Timer : public Object {
public:
        Timer();
        ~Timer();

        void start( unsigned milliseconds, unsigned vector );
        void stop();
        int idle();
};

#endif Timer_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 26 05:05 1987  ../h/Universe.h Page 1


/*
 * Universe.h: Universe class description. A Universe is all the Spaces a
 *             processor can access at any one time.
 *
 *      $Header: Universe.h,v 11.0 87/05/21 15:50:25 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Universe.h,v $
 * Revision 11.0  87/05/21  15:50:25  russo
 * Console input and private stores.
 *
 * Revision 10.3  87/05/06  19:27:27  russo
 * added loadContextFor() method.
 *
 * Revision 10.2  87/04/22  20:38:05  russo
 * added spaceContaining method
 *
 * Revision 10.1  87/04/22  09:39:41  russo
 * renames spaces
 *
 * Revision 10.0  87/04/22  07:33:57  russo
 * New Spaces. Universes and CPU objects work, Finally!
 *
 * Revision 9.1  87/04/11  19:45:25  russo
 * initial revision.
 */

#include "Object.h"     // parent class
#include "VM.h"
#include "Store.h"
#include "Space.h"

#ifndef Universe_h
#define Universe_h

/*
 * For now the Universe has a fixed amount of spaces that can be mapped in
 * to fixed places. This will be extended when I have time.
 */
class Universe : public Object {
        PTE * firstLevelPageTable;
        Space * kernelSpace;
        Space * userSpace;
public:
        Universe( Universe * where, PTE * pageTable );
        ~Universe();
        void addSpace( Space * aSpace );
        Space * spaceContaining( void * address );
        void loadContextFor( Thread * newThread );
};

#endif Universe_h
```

```
/*
 * Space.h: Space class description.
 *
 *      $Header  Space.h v 11.7 87/05/26 21:41:33 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   Space.h,v $
 * Revision 11.7  87/05/26  21:41:33  russo
 * ???
 *
 * Revision 11.6  87/05/26  05:04:50  russo
 * changed name of allocatePointerTable argument
 *
 * Revision 11.5  87/05/25  17:36:25  russo
 * sorry, not implemented.
 *
 * Revision 11.3  87/05/24  23:13:38  russo
 * more work on new allocation stuff
 *
 * Revision 11.2  87/05/24  16:46:27  russo
 * redid allocation stuff and other private methods.
 *
 * Revision 11.1  87/05/24  05:18:57  russo
 * added new allocate method definition
 *
 * Revision 11.0  87/05/21  15:50:07  russo
 * Console input and private stores.
 *
 * Revision 10.10  87/05/14  19:57:05  russo
 * added isIn() method for use to check wheather an address is managed by
 * a space
 *
 * Revision 10.8  87/04/22  16:56:14  russo
 * fixed KernelSpace constructor args.
 *
 * Revision 10.2  87/04/22  16:13:38  russo
 * switch constructor to take base and length rather than start and end
 *
 * Revision 10.1  87/04/22  09:36:41  russo
 * renaming from NewSpace
 *
 * Revision 10.0  87/04/22  07:33:35  russo
 * New Spaces, Universes and CPU objects [illegible] Finally!
 *
 * Revision 9.5  87/04/14  20:58:46  russo
 * created KernelSpace subclass
 *
 * Revision 9.1  87/04/13  04:50:24  russo
 * initial revision. (actually a rewrite of the old stuff)
 */

#ifndef Space_h
#define Space_h
```

```
#include "Object.h"     // parent class
#include "Lock.h"
#include "VM.h"
#include "Store.h"
#include "FaultHandler.h"

/*
 * Space - base class.
 */

enum allocationType { prefetch, faultIn };

class Space : public Object {

        friend class Universe;

protected:
        Lock lock;
        Store * store;
        void * baseAddress;
        int length;

        unsigned vTopPage;
        struct {
                PTE firstLevelPTE;
                PTE * secondLevelPTE;
        } table[256];
        FaultHandler * faultHandler[ MAXHANDLERS ];

        virtual void getPointerTables( unsigned lowPage, unsigned highPage );

        virtual void buildMappings( unsigned int start, unsigned int count,
                                    FaultHandler * handler );

        virtual PTE * allocatePointerTable( int page );
        virtual int convertFaultHandlerToIndex( FaultHandler * handler );
public:
        Space( Store * store, void * baseAddress, int length );
        ~Space();

        virtual void * startAddress();
        virtual void * endAddress();
        virtual int isIn( void * vaddr );
        virtual int isValid( void * vaddr );

        virtual void * allocate( unsigned int count, FaultHandler * handler,
                                 allocationType type );
        virtual void * allocate( void * base, unsigned int count,
                                 FaultHandler * handler, allocationType type );

        virtual void map( void * page, void * frame );

        FaultHandler * handler( void * vaddr );

/* these are dead */
        virtual void * allocate( unsigned int pageCount );
        virtual void map( void * vbase, void * pbase, unsigned int pageCount );
};

class KernelSpace : public Space {

private:
        void getPointerTables( unsigned lowPage, unsigned highPage );

        virtual PTE * allocatePointerTable( int page );
public:
        KernelSpace( Store *, void * baseAddress, int length );
        ~KernelSpace();
};

#endif Space_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * Exception.h: event class description.
 *
 *      $Header: Exception.h,v 11.0 87/05/21 15:49:35 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Exception.h,v $
 * Revision 11.0  87/05/21  15:49:35  russo
 * Console input and private stores.
 *
 * Revision 10.13  87/05/01  11:02:06  russo
 * renamed from Event.h
 */

#ifndef Exception_h
#define Exception_h

#include "Object.h"     // parent class
#include "Thread.h"
#include "Frame.h"

typedef void ( * HandlerFunction )( struct Frame * frame );

class Exception : public Object {
protected:
public:
        virtual void post( struct Frame * frame );
};

class SystemException : public Exception {
protected:
        HandlerFunction handler;
public:
        SystemException( HandlerFunction theHandler );
        ~SystemException();
        void post( struct Frame * frame );
};

class InterruptException : public Exception {
protected:
        Thread * awaiter;
public:
        InterruptException();
        ~InterruptException();
        void post( struct Frame * frame );
        void await();
};

#endif Exception_h
```

```
/*
 * SpaceList.h: SpaceList class description. A SpaceList is an ordered
 *              list of Spaces. It is specially coded and not a sub-
 *              class of some more abstract list class since it it very
 *              heavily used and high efficiency is desired.
 *
 *      $Header: SpaceList.h,v 11.1 87/05/23 20:19:26 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   SpaceList.h,v $
 * Revision 11.1  87/05/23  20:18:26  russo
 * added enclosing ifdef
 * ...
 *
 * Revision 11.0  87/05/21  15:50:10  russo
 * Console input and private stores.
 *
 * Revision 10.2  87/04/23  21:16:22  russo
 * made Object the parent class for pedantic reasons.
 *
 * Revision 10.1  87/04/23  21:15:33  russo
 * initial revision.
 */

#ifndef SpaceList_h
#define SpaceList_h

#include "Object.h"
#include "Space.h"

class spaceListNode : public Object {
public:
        spaceListNode * next;
        spaceListNode * last;
        Space * data;

        spaceListNode( Space * s ) {
                next = 0;
                last = 0;
                data = s;
        }
};

class SpaceList : public Object {
protected:
        spaceListNode * head;
        spaceListNode * tail;
        spaceListNode * iterator;
        Lock lock;
public:
        SpaceList();
        ~SpaceList();
        virtual void add( Space * space );
        virtual int remove( Space * space );
```

```
        virtual int inquire( Space * space );

        virtual void startIteration();
        virtual Space * iterateNext();
};

#endif SpaceList_h
```

```
May 26 05:05 1987  ../h/Frame.h Page 1


/*
 * Frame.h: description of the frame on the interrupt stack passed to
 *              exception handlers.
 *
 *      $Header: Frame.h,v 11.0 87/05/21 15:49:50 russo Exp $
 *      $Locker:  $
 *
 * Modification History:
 *      $Log:   Frame.h,v $
 * Revision 11.0  87/05/21  15:49:50  russo
 * Console input and private stores.
 *
 * Revision 10.7  87/04/30  14:55:15  russo
 * added usp to frame.
 *
 * Revision 10.6  87/04/29  07.06.13  russo
 * made all fields unsigned int(shot)
 *
 * Revision 10.1  87/04/27  18:39:03  russo
 * initial revision.
 */

#ifndef Frame_h
#define Frame_h

struct Frame {
        unsigned int    sp;
        unsigned int    fp;
        unsigned int    r7;
        unsigned int    r6;
        unsigned int    r5;
        unsigned int    r4;
        unsigned int    r3;
        unsigned int    r2;
        unsigned int    r1;
        unsigned int    r0;
        unsigned int    vectorNumber;
        unsigned int    pc
        unsigned short  mod;    /* this implicitly determines the value of the
                                   sb register. so we dont save sb. */
        unsigned short  psr;
};

#endif Frame_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 26 05:05 1987  ../h/Vectors.h Page 1


/*
 * Vectors.h: vector numbers for processor traps/interrupts.
 *
 *      $Header: Vectors.h,v 11.0 87/05/21 15:50:35 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Vectors.h,v $
 * Revision 11.0  87/05/21  15:50:35  russo
 * Console input and private stores.
 *
 * Revision 10.5  87/05/13  13:47:59  johnston
 * Added definition of CPUCLASS_MAX for assertions, etc.
 *
 * Revision 10.4  87/05/13  13:45:49  johnston
 * Changed CPUCLASS to be CPUCLASS_HALTED (0) and CPUCLASS_RUNNING (1).
 * Did this so that a halted CPU won't get class interrupts that will
 * never get serviced.
 *
 * Revision 10.3  87/04/30  16:25:26  johnston
 * Added CPUCLASS definition.
 *
 * Revision 10.1  87/04/28  09:28:06  russo
 * initial revision
 */

#ifndef Vectors_h
#define Vectors_h

/*
 * Trap Vectors
 */
#define NVI_Vector      0               /* Non-Vectored Interrupt */
#define NMI_Vector      1               /* Non-Maskable Interrupt */
#define ABT_Vector      2               /* Abort (VM Error) Trap */
#define FPU_Vector      3               /* Floating Point Exception Trap */
#define ILL_Vector      4               /* Illegal Instruction Trap */
#define SVC_Vector      5               /* Supervisor Call Instruction Trap */
#define DVZ_Vector      6               /* Divide by Zero Trap */
#define FLG_Vector      7               /* Trap on Flag */
#define BPT_Vector      8               /* Breakpoint Trap */
#define TRC_Vector      9               /* Trace Trap */
#define UND_Vector      10              /* Undefined Instruction Trap */
#define RESERVED_11_Vector      11      /* Reserved */
#define RESERVED_12_Vector      12      /* Reserved */
#define RESERVED_13_Vector      13      /* Reserved */
#define RESERVED_14_Vector      14      /* Reserved */
#define RESERVED_15_Vector      15      /* Reserved */

/*
 * Interrupt Vectors
 */
#define TIMESLICE_Vector        16      /* Time Slice Counter Interrupt */
#define CONSOLE_Vector          17      /* Console Input Interrupt */
```

```
#define UNUSED_18_Vector        18
#define UNUSED_19_Vector        19
#define UNUSED_20_Vector        20

/*
 * Vector bus classes.
 */
#define CPUCLASS_HALTED         0       /* Halted CPU class */
#define CPUCLASS_RUNNING        1       /* Normal  running CPU class */
#define CPUCLASS_MAX            1       /* Highest class used */

#endif Vectors_h
```

```
/*
 * Scheduler.h: a dispatch queue description.
 *
 *      $Header: Scheduler.h,v 11.0 87/05/21 15:50:04 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Scheduler.h,v $
 * Revision 11.0  87/05/21  15:50:04  russo
 * Console input and private stores.
 *
 * Revision 10.2  87/05/12  17:44:04  russo
 * added enclosing #ifdef
 *
 * Revision 10.1  87/05/04  17:21:20  russo
 * initial revision
 */

#ifndef Scheduler_h
#define Scheduler_h

#include "Thread.h"

class Scheduler {

protected:
        Lock lock;
        int nextIn;
        int nextOut;
        int maxThreads;
        Thread ** Queue;

public:
        Scheduler( int maxThreads );
        ~Scheduler();
        void add( Thread * );
        Thread * removeNext();

};

#endif Scheduler_h
```

```
/*
 * PrivateMemory.h  Inline functions and constants to calculate the offsets
 * of various things into the private virtual memory area
 * BootPhysical() and BootVirtual() use these to cooperate on where things go.
 *
 *      $Header: PrivateMemory.h,v 11.2 87/05/26 06:46:01 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   PrivateMemory.h,v $
 * Revision 11.2  87/05/26  06:46:01  russo
 * make stack 2 pages large
 *
 * Revision 11.1  87/05/23  20:17:42  russo
 * added surrounding ifdef.
 *
 * Revision 11.0  87/05/21  15:49:59  russo
 * Console input and private stores.
 *
 * Revision 10.2  87/05/14  17:47:23  russo
 * added an extra const for the second page table page
 *
 * Revision 10.1  87/05/14  17:26:41  russo
 * initial revision.
 */

#ifndef PrivateMemory_h
#define PrivateMemory_h

#include "Store.h"
#include "CPU.h"
#include "VM.h"
#include "Universe.h"
#include "Thread.h"

/*
 * VM Map of the per-CPU private memory area.
 *
 *      +-----------------------------------+ <<-- End of private VM.
 *      |                                   |
 *      | As many pages as needed to store  |
 *      | the rest of the private store     |
 *      | state information. The of memory  |
 *      | up to the end of the per-CPU      |
 *      | private area is available.        |
 *      |                                   |
 *      | If more things need added to this |
 *      | area, add them below the private  |
 *      | store and bump it up higher       |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 10 pages.
 *      |                                   |
 *      | The beginning of the private      |
 *      | store state information           |
 *      |                                   |
```

```
 *      +-----------------------------------+ <<-- Base + 9 pages.
 *      |                                   |
 *      | Intentionally unmapped to catch   |
 *      |    stack underflow                |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 8 pages.
 *      |                                   |
 *      | The Germ Threads stack page 2     |
 *      |                                   |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 7 pages.
 *      |                                   |
 *      | The Germ Threads stack page 1     |
 *      |                                   |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 6 pages.
 *      |                                   |
 *      | Intentionally unmapped to catch   |
 *      |    stack overflow                 |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 5 pages.
 *      |                                   |
 *      | The Germ Thread objects state     |
 *      |        information                |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 4 pages.
 *      |                                   |
 *      | Second page of the CPUs           |
 *      |        page table                 |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 3 pages.
 *      |                                   |
 *      | First page of the CPUs            |
 *      |        page table                 |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 2 pages.
 *      |                                   |
 *      | Pointer table to point to all     |
 *      |     of the stuff here             |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base + 1 page.
 *      |                                   |
 *      | CPU and Universe objects state    |
 *      |         information               |
 *      |                                   |
 *      +-----------------------------------+ <<-- Base of private VM area
 */

/*
 * The CPU object.
 */
const int mePage = 0;
static inline CPU * meLocation( char * base )
{
        return( (CPU *) ( base + 0 ) );
}
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * The Universe object.
 */
const int universePage = 0;
static inline Universe * universeLocation( char * base )
{
        return( (Universe *) ( base + sizeof( CPU ) ) );
}

/*
 * The pointer table to access private things with.
 */
const int privatePointerTablePage = 1;
static inline PTE * privatePointerTableLocation( char * base )
{
        return( (PTE *) ( base + PAGESIZE ) );
}

/*
 * This processors page table.
 */
const int pageTablePage = 2;
const int secondPageTablePage = 3;
static inline PTE * pageTableLocation( char * base )
{
        return( (PTE *) ( base + pageTablePage*PAGESIZE ) );
}

/*
 * The initial thread.
 */
const int germThreadPage = 4;
static inline GermThread * germThreadLocation( char * base )
{
        return( (GermThread *) ( base + germThreadPage*PAGESIZE ) );
}

/*
 * The stack for the initial thread
 */
const int stackPage = 6;
static inline char * stackLocation( char * base )
{
        return( (char *) ( base + stackPage*PAGESIZE ) );
}

/*
 * The private Store state information
 */
const int privateStorePage = 9;
static inline Store * privateStoreLocation( char * base )
{
        return( (Store *) ( base + privateStorePage * PAGESIZE ) );
}
```

```
#endif PrivateMemory_h
```

ORIGINAL PAGE IS
OF POOR QUALITY

Owner	russo at m.cs.uiuc.edu
Name	KernelMain.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:22:50 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 3
Pages printed: 3

Paper size (width, height):
  2560, 3328
Document length:
  9776 bytes
```

```
May 28 10:02 1987  KernelMain.c Page 1


/*
 * KernelMain():
 *      This routine sets up things that need initialized before
 *      the processors are all let loose.
 *
 *      $Header: KernelMain.c,v 11.4 87/05/25 05:17:33 russo Exp $
 *      $Locker:  $
 */
/*
 * Revision History:
 *      $Log:   KernelMain.c,v $
 * Revision 11.4  87/05/25  05:17:33  russo
 * only do dselective debug printing
 *
 * Revision 11.3  87/05/24  06:13:56  russo
 * took out VERY annoying debug
 *
 * Revision 11.1  87/05/21  16:51:10  russo
 * switch ways of deleteing threads
 *
 * Revision 11.0  87/05/21  15:54:43  russo
 * Console input and private stores.
 *
 * Revision 10.32  87/05/17  14:04:26  russo
 * keep deleteQueue as a per-CPU member
 *
 * Revision 10.24  87/05/13  20:29:56  russo
 * split out the time slice thread code and did some other rearrangements.
 *
 * Revision 10.20  87/05/13  06:07:23  johnston
 * Added creation of console thread.
 *
 * Revision 10.17  87/05/12  17:35:26  russo
 * restructured to be run by the germs initial thread. Also sets us up
 * to use the new scheduler() member of the CPU class.
 *
 * Revision 10.0  87/04/22  07:38:16  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:12:29  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:33:45  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:49:34  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 1.1  87/02/23  23:34:11  russo
 * Initial revision
 */

#include "Assert.h"
#include "Debug.h"
#include "md_tuneable.h"
#include "Store.h"
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 28 10:02 1987  KernelMain.c Page 2


#include "File.h"
#include "FaultHandler.h"
#include "Space.h"
#include "Task.h"
#include "Scheduler.h"
#include "Exception.h"
#include "Lock.h"
#include "Universe.h"
#include "CPU.h"
#include "Vectors.h"

/*
 * Global Exception handlers.
 */
extern void ABTTrap( struct Frame * );
SystemException VMException( ABTTrap );

extern void SVCTrap( struct Frame * );
SystemException SVCException( SVCTrap );

InterruptException ConsoleInterruptException;

/*
 * Miscellaneous variables.
 */
static int setupDone = 0;
static Lock setupLock;
static Scheduler * runQ;        // run queue for this kernel

static int ConsoleThreadBuilt = 0;
static Lock ConsoleThreadLock;

typedef void (* KPFV)();

void
KernelMain( unsigned int ID )
{
        /*
         * Initial sanity Assertions and Debugging.
         */
        Assert( Me != 0 );
        Debug( "Processor %x has joined the kernel (ID=%x)\n", Me->id(), ID );
        Assert( ID == Me->id() );
        Assert( Me->universe() != 0 );
        Assert( Me->heapSpace() != 0 );
        Assert( Me->currentThread() != 0 );
        Assert( Me->idleThread() != 0 );
        Assert( Me->currentThread() == Me->idleThread() );
        Assert( Me->scheduler() == 0 );
        Assert( Me->threadToDelete() == 0 );
        Assert( Me->privateStore() != 0 );
        Assert( Me->globalStore() != 0 );

        extern int InterruptsDisabled();
        Assert( InterruptsDisabled() );
	/*
	 * Install the global (common amoung all CPUs executing this kernel)
	 * exception handlers. The ABT handler is especially important since
	 * it will allow page faults to occur and be handled.
	 */
	Debug( "KernelMain: Installing global exceptions\n" );
	Me->setException( ABT_Vector, &VMException );
	Me->setException( SVC_Vector, &SVCException );
	Me->setException( CONSOLE_Vector, &ConsoleInterruptException );

	/*
	 * Install the local (private) exception handlers.
	 *
	 * Build a stack for the clock thread, then construct and dispatch it.
	 * It will build, install and await the time slice exception.
	 */
	Debug( "KernelMain: Building kernel clock thread\n" );
	extern void SwitchTo( Thread * );
	extern void * KernelThreadStackAllocator( int );
	extern void Ticker( int );

	int * stack = (int *) KernelThreadStackAllocator( 1 );
	Assert( stack != 0 );

	Thread * clock = new InterruptThread( (PFV) Ticker, stack, 0, 0, 0 );
	Debug( "KernelMain: clock thread %x (stack=%x)\n", clock, stack );
	Assert( clock != 0 );

	SwitchTo( clock );
	Debug( "KernelMain: clock dispatch returned\n" );
	Assert( Me->id() == ID );

	/*
	 * Build a stack for the console thread, then construct and dispatch it.
	 */
	ConsoleThreadLock.acquire();
	if( ! ConsoleThreadBuilt ) {
		/*
		 * If were the one going to build it, indicate so and let
		 * everyone else proceede since there is no reason for them
		 * to wait for us.
		 */
		ConsoleThreadBuilt = 1;
		ConsoleThreadLock.release();

		Debug( "KernelMain: Building console thread\n" );

		stack = (int *) KernelThreadStackAllocator( 1 );
		Assert( stack != 0 );

		extern void ConsoleThreadEntry( int );
		Thread * console = new InterruptThread(
			(PFV) ConsoleThreadEntry, stack, 0, 0, 0 );
		Debug( "KernelMain: console thread %x (stack=%x)\n", console,
			stack );
		Assert( console != 0 );
```

```
		SwitchTo( console );
		Debug( "KernelMain: console dispatch returned\n" );
		Assert( Me->id() == ID );
	}
	else {
		ConsoleThreadLock.release();
	}

	/*
	 * Check to see if the setup portion of the kernel has been done by
	 * someone else, and if not do it. Otherwise wait for them to finish
	 * then continue on.
	 */
	setupLock.acquire();
	if( ! setupDone ) {

		Debug( "KernelMan: Doing kernel setup\n" );
		setupDone = 1;

		/*
		 * Initialize the run queue scheduler.
		 */
		runQ = new Scheduler( 100 ); // hold 100 threads. DEFINE THIS!!!
		Debug( "KernelMain: runQ = %x\n", runQ );
		Assert( runQ != 0 );

		/*
		 * Create the initial Space.
		 */
		CPUPrintf( "KernelMain: Creating init space from %x\n",
			Me->globalStore() );
		Space * initialSpace = new Space( Me->globalStore(),
			(void *) TASKLOWADDR, TASKHIGHADDR - TASKLOWADDR );
		Debug( "KernelMain: initialSpace = %x\n", initialSpace );
		Assert( initialSpace != 0 );

		/*
		 * Create the File to load the space from.
		 */
		extern char InitialCode[];
		extern int InitialCodeSize;

		Debug( "KernelMain: Creating the initial space's COFF file\n" );
		File * initialFile = new MemoryFile( InitialCode,
			InitialCodeSize );
		Debug( "KernelMain: initialFile = %x\n", initialFile );
		Assert( initialFile != 0 );

		/*
		 * Initialize the space from the COFF image of the initial Task
		 * in the file created above.
		 */
		extern KPFV SetupSpaceFromCOFFImage( Space *, File * );

		Debug( "KernelMain: initializing space from the COFF image\n" );
```

May 28 10:02 1987 KernelMain.c Page 5

```
                KPFV entryPoint = SetupSpaceFromCOFFImage( initialSpace,
                        initialFile );
                Debug( "KernelMain: entryPoint = %x\n", entryPoint );
                Assert( entryPoint != 0 );

                /*
                 * Create a task int the initial space.
                 */
                Debug( "KernelMain: Creating initial task(%x, %x)\n",
                        initialSpace, entryPoint );
                Task * initialTask = new Task( initialSpace,
                        (KPFV) entryPoint );
                Debug( "KernelMain: initialTask = %x\n", initialTask );
                Assert( initialTask != 0 );

                /*
                 * Enqueue the tasks initial thread on the scheduler we
                 * made above.
                 */
                Debug( "KernelMain: adding initialThread %x to runQ\n",
                        initialTask->initialThread() );
                Assert( initialTask->initialThread() != 0 );
                runQ->add( initialTask->initialThread() );
        }
        setupLock.release();

        /*
         * Now that all the initial interrupt threads have been dispatched
         * and started, turn on interrupts, set the scheduler member of the
         * CPU object to the kernels scheduler and begin round robin scheduling.
         */
        extern void EnableInterrupts();

        Debug( "KernelMain: enabling interrupts\n" );
        EnableInterrupts();
        Me->setScheduler( runQ );

        /*
         * Keep looping removing things from the scheduler and dispatching
         * them.
         */
        Debug( "KernelMain: entering idle loop\n" );
        while( 1 ) {
                extern int InterruptsEnabled();
                Assert( InterruptsEnabled() );

                Thread * aThread;
                Assert( Me->scheduler() != 0 );
                Assert( Me->threadToDelete() == 0 );
                while( ( aThread = Me->scheduler()->removeNext() ) == 0 );

                CPUPrintf( "Idle(): aThread = %x\n", aThread );
                SwitchTo( aThread );
                CPUPrintf( "Idle(): RETURNED, Checking for Threads to delete\n" );
                Assert( Me->id() == ID );
                Assert( Me->currentThread() == Me->idleThread() );
```

ORIGINAL PAGE IS
OF POOR QUALITY

May 28 10:02 1987 KernelMain.c Page 6

```
                /*
                 * Check to see if there is a Thread to delete.
                 */
                if( ( aThread = Me->threadToDelete() ) != 0 ) {
                        CPUPrintf( "KernelMain: deleting %x\n", aThread );
                        delete aThread;
                        Me->setThreadToDelete( 0 );
                }
        }
}

/*
 * The rest of this file is only here for testing.
 * It belongs in, and will be moved to, somewhere else in the kernel when
 * it is completed.
 */

/*
 * Build a stack for a kernel (interrupt) thread. The stack is allocated
 * virtually in the processors current heap space and physically from
 * Me->globalStore().The pointer returned is to the start (bottom) of the stack.
 * This is actually the high end of the memory region allocated.
 *
 * FIX THIS TO ALLOCATE "SANDWICH" VIRTUAL PAGES TO CATCH OVER/UNDERFLOW!
 * AND TO USE THE PROPER STORE
 */
void *
KernelThreadStackAllocator( int numberOfPages )
{
        Debug( "KernelStackAllocator( %d )\n", numberOfPages );
        Assert( Me->globalStore() != 0 );
        Assert( Me->heapSpace() != 0 );

        char * vstack = (char *) Me->heapSpace()->allocate( numberOfPages );
        void * pstack = Me->globalStore()->allocate( numberOfPages );
        Assert( vstack != 0 );
        Assert( pstack != 0 );
        Me->heapSpace()->map( vstack, pstack, numberOfPages );

        return( vstack + (numberOfPages<<PAGESHIFT) );
}
```

Owner	russo at m.cs.uiuc.edu
Name	Store.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:33:39 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 2
Pages printed: 2

Paper size (width, height):
  2560, 3328
Document length:
  5930 bytes
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * Store.c - Physical memory allocation.
 *
 *      $Header: Store.c,v 11.0 87/05/21 15:57:18 russo Exp $
 *      $Locker:  $
 */
/*
 * Revision History:
 *      $Log:   Store.c,v $
 * Revision 11.0  87/05/21  15:57:18  russo
 * Console input and private stores.
 *
 * Revision 10.5  87/05/15  05:55:58  johnston
 * Fixed calculation of Store state page count in constructor.
 *
 * Revision 10.3  87/05/14  21:56:09  johnston
 * Hopefuly fixed some stuff.  Changed the interface, too.
 * Gotta hunt around and find all the places I broke it.
 *
 * Revision 10.0  87/04/22  07:43:34  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:17:36  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:37:44  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:52:55  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 1.1  87/02/23  18:20:25  russo
 * Initial revision
 */

#include "md_constants.h"
#include "Assert.h"
#include "Debug.h"
#include "Lock.h"
#include "Store.h"
#include "VM.h"

Store::Store(unsigned basePage, unsigned pageCount, unsigned * stateBasePage )
{
        /*
         * Debugging and entry assertions.
         */
        Debug( "Store::Store(%x,%x,%x)\n", basePage, pageCount, stateBasePage );
        Assert( stateBasePage );
        Debug( "Store::Store: *stateBasePage: %x\n", *stateBasePage );
        Assert( pageCount != 0 );
        Assert( this == 0 );

        /*
         * Figure out our sizes, etc.
         */
```

```
        unsigned pagesPerSetEntry = 8 * sizeof( unsigned );
        unsigned setEntryCount = ( pageCount + ( pagesPerSetEntry - 1 ) ) /
                                 pagesPerSetEntry;
        Assert( ( setEntryCount * pagesPerSetEntry ) >= pageCount );
        unsigned stateByteCount = ( sizeof( Store ) - sizeof( unsigned ) ) +
                                  ( setEntryCount * sizeof( unsigned ) );
        stateByteCount = PageCeiling( stateByteCount );
        unsigned statePageCount = addrToPage( (char *) stateByteCount );

        /*
         * Allocate ourself.
         */
        this = (Store *) ( *stateBasePage * PAGESIZE );
        *stateBasePage += statePageCount;
        Debug ( "Store::Store: this: %x (%d pages)\n", this, statePageCount );

        /*
         * Initialize the Store state information.
         */
        this->basePage = basePage;
        this->highPage = basePage + pageCount;
        this->freePageCount = pageCount;
        this->setEntryCount = setEntryCount;
        this->pagesPerSetEntry = pagesPerSetEntry;
        for ( unsigned i = 0; i < this->setEntryCount; i++ )
                this->set[ i ] = 0;
        Debug( "Store::Store: basePage: %x, highPage: %x, freePageCount: %x\n",
                this->basePage, this->highPage, this->freePageCount );
        Debug( "Store::Store: setEntryCount: %x, pagesPerSetEntry: %x\n",
                this->setEntryCount, this->pagesPerSetEntry );
}

Store::~Store ()
{
        extern void Halt();

        this->lock.acquire();
        PanicPrintf( "Store::~Store: DESTRUCTOR CALLED on %x\n", this );
        this = 0;
        Halt();
}

/*
 * Store::nextFree
 *
 * Return the first free page after or including the specified page.
 * Return this->highPage if none were found.
 */

unsigned
Store::nextFree( unsigned page )
{
        Assert( page >= this->basePage );
        Assert( page < this->highPage );
        while ( page < this->highPage ) {
                if ( ! this->marked( page ) )
May 28 10:02 1987  Store.c Page 3


                        return  page ;
                page++;
        }
        return( this->highPage );
}

/*
 * Store::contiguous
 *
 * Return the number of contiguous free pages starting at the specified page.
 * Look for no more than the number of pages specified.
 */

unsigned
Store::contiguous( unsigned page, unsigned pageCount )
{
        Assert( this->inStore( page ) );
        for ( unsigned count = 0; count < pageCount; count++, page++ ) {
                if ( page == this->highPage )
                        return ( count );
                if ( this->marked( page ) )
                        return ( count );
        }
        Assert( this->inStore( page ) );
        return ( count );
}

char *
Store::allocate( unsigned pageCount )
{
        /*
         * First fit search for number of contiguous pages requested.
         */
        Debug( "Store::allocate( %d ): this = %x\n", pageCount, this );
        Assert( pageCount != 0 );
        this->lock.acquire();
        unsigned basePage = this->basePage;
        unsigned freePageCount = 0;
        while ( basePage < this->highPage ) {
                basePage = this->nextFree( basePage );
                freePageCount = this->contiguous( basePage, pageCount );
                if ( freePageCount == pageCount ) {
                        Assert( this->inStore( basePage ) );
                        unsigned page = basePage;
                        for ( unsigned i = 0; i < freePageCount; i++, page++ )
                                this->mark( page );
                        char * addr = (char *) pageToAddr( basePage );
                        Assert( this->inStore( addr ) );
                        this->lock.release();
                        Debug( "Store::allocate: %x\n", addr );
                        return ( addr );
                }
                basePage += freePageCount;
        }
        this->lock.release();
        CPUPrintf( "Store::allocate: FAILED (this = %x)\n", this );
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 28 10:02 1987  Store.c Page 4

        return ( 0 );
}

void
Store::deallocate( char * baseAddr, unsigned pageCount )
{
        Debug( "Store::deallocate(%x,%d): this = %x\n",
                baseAddr, pageCount, this );
        Assert( ( (unsigned) baseAddr % PAGESIZE ) == 0 );
        unsigned basePage = addrToPage( baseAddr );
        Assert( this->inStore( basePage ) );
        Assert( this->inStore( ( basePage + pageCount ) - 1 ) );
        this->lock.acquire();
        while ( pageCount-- ) {
                this->unmark( basePage );
                basePage++;
        }
        this->lock.release();
}

void
Store::reserve( unsigned basePage, unsigned pageCount )
{
        Debug( "Store::reserve(%x,%d): this = %x\n",
                basePage, pageCount, this );
        Assert( this->inStore( basePage ) );
        Assert( pageCount != 0 );
        Assert( this->inStore( ( basePage + pageCount ) - 1 ) );
        this->lock.acquire();
        while ( pageCount-- ) {
                this->mark( basePage );
                basePage++;
        }
        this->lock.release();
}
```

Owner	russo at m.cs.uiuc.edu
Name	Space.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:36:24 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 7
Pages printed: 7

Paper size (width, height):
  2560, 3328
Document length:
  19756 bytes
```

```
May 28 10:03 1987  Space.c Page 1


/*
 * Space.c  Space class implementation.
 *
 *      $Header: Space.c,v 11.11 87/05/26 07:08:41 johnston Exp $
 *      $Locker: russo $
 */
/*
 * Modification History:
 *      $Log:   Space.c,v $
 * Revision 11.11  87/05/26  07:08:41  johnston
 * Debugging off.
 *
 * Revision 11.10  87/05/26  05:03:28  russo
 * changed argument to allocatePointerTable to be the page not the l1Index
 *
 * Revision 11.9  87/05/25  17:37:35  russo
 * sorry, not implemented.
 *
 * Revision 11.8  87/05/25  17:33:58  russo
 * updating the rest of the methods.
 *
 * Revision 11.6  87/05/25  05:40:30  russo
 * more debugging to track down a panic
 *
 * Revision 11.4  87/05/24  17:06:22  russo
 * added missing KernelSpace method.
 *
 * Revision 11.2  87/05/24  16:47:18  russo
 * finishing up new allocation stuff
 *
 * Revision 11.0  87/05/21  15:57:09  russo
 * Console input and private stores.
 *
 * Revision 10.21  87/05/15  05:14:30  johnston
 * Fixed Store->deallocate parameters.
 *
 * Revision 10.20  87/05/14  20:05:45  russo
 * added isIn method to space class to test if an address is managed by the
 * space.
 *
 * Revision 10.8  87/04/22  21:11:33  russo
 * leave FaultHandler[0] empty so un-handled pages return a 0 fault handler.
 *
 * Revision 10.0  87/04/22  07:43:27  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.1  87/04/13  04:50:57  russo
 * initial revision.
 */

#include "Debug.h"
#include "Assert.h"
#include "VM.h"
#include "Store.h"
#include "Space.h"
#include "FaultHandler.h"
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 28 10:03 1987  Space.c Page 2

#include "CPU.h"

/*
 * Space constructor.
 *
 *   Arguments:
 *       store:  used to allocate pyhsical memory for the pointer table state
 *               information from.
 *       base:   the starting address of the space.
 *       length: the length of the space.
 */

Space::Space( Store * store, void * base, int length )
{
        /*
         * Entry debugging and assertions.
         * (Check for proper alignments of the Space boundries.)
         */
        Debug( "Space::Space( store:%x base:%x, length:%x): this = %x\n",
                store, base, length, this );
        Assert( this != 0 );
        Assert( store != 0 );
        Assert( ( (unsigned) base & 0xffff ) == 0 );
        Assert( ( (unsigned)( (char *) base + length ) & 0xffff ) == 0 );

        /*
         * Set the space boundary instance variables.
         */
        this->baseAddress = base;
        this->length = length;
        this->vTopPage = addrToPage( base );

        /*
         * Initialize the pointer table information kept for the space.
         */
        for( int i = 0; i < 256; i++ ) {
                this->table[i].secondLevelPTE = 0;
                this->table[i].firstLevelPTE = (PTE) 0;
        }
        this->store = store;

        /*
         * Initialize the spaces fault handler table.
         */
        for( i = 0; i < MAXHANDLERS; i++ ) {
                this->faultHandler[i] = 0;
        }
}

/*
 * Space destructor. Boy does this need work!
 */
Space::~Space()
{
        CPUPrintf( "Space::~Space: this=%x\n", this );
        Assert( ! this->lock.held() );
```

```
/*
 * Space.c: Space class implementation.
 *
 *      $Header: Space.c,v 11.11 87/05/26 07:08:41 johnston Exp $
 *      $Locker: russo $
 */
/*
 * Modification History:
 *      $Log:   Space.c,v $
 * Revision 11.11  87/05/26  07:08:41  johnston
 * Debugging off.
 *
 * Revision 11.10  87/05/26  05:03:28  russo
 * changed argument to allocatePointerTable to be the page not the l1Index
 *
 * Revision 11.9  87/05/25  17:37:35  russo
 * sorry, not implemented.
 *
 * Revision 11.8  87/05/25  17:33:58  russo
 * updating the rest of the methods.
 *
 * Revision 11.6  87/05/25  05:40:33  russo
 * more debugging to track down a panic
 *
 * Revision 11.4  87/05/24  17:06:22  russo
 * added missing KernelSpace method.
 *
 * Revision 11.2  87/05/24  16:47:18  russo
 * finishing up new allocation stuff
 *
 * Revision 11.0  87/05/21  15:57:09  russo
 * Console input and private stores.
 *
 * Revision 10.21  87/05/15  05:14:30  johnston
 * Fixed Store->deallocate parameters.
 *
 * Revision 10.20  87/05/14  20:05:45  russo
 * added isIn method to space class to test if an address is managed by the
 * space.
 *
 * Revision 10.8  87/04/22  21:11:33  russo
 * leave FaultHandler[0] empty so un-handled pages return a 0 fault handler.
 *
 * Revision 10.0  87/04/22  07:43:27  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.1  87/04/13  04:50:57  russo
 * initial revision.
 */

#include "Debug.h"
#include "Assert.h"
#include "VM.h"
#include "Store.h"
#include "Space.h"
#include "FaultHandler.h"
```

```
#include "CPU.h"

/*
 * Space constructor.
 *
 *   Arguments:
 *       store:  used to allocate pyhsical memory for the pointer table state
 *               information from.
 *       base:   the starting address of the space.
 *       length: the length of the space.
 */

Space::Space( Store * store, void * base, int length )
{
        /*
         * Entry debugging and assertions.
         * (Check for proper alignments of the Space boundries.)
         */
        Debug( "Space::Space( store:%x base:%x, length:%x): this = %x\n",
                store, base, length, this );
        Assert( this != 0 );
        Assert( store != 0 );
        Assert( ( (unsigned) base & 0xffff ) == 0 );
        Assert( ( (unsigned)( (char *) base + length ) & 0xffff ) == 0 );

        /*
         * Set the space boundary instance variables.
         */
        this->baseAddress = base;
        this->length = length;
        this->vTopPage = addrToPage( base );

        /*
         * Initialize the pointer table information kept for the space.
         */
        for( int i = 0; i < 256; i++ ) {
                this->table[i].secondLevelPTE = 0;
                this->table[i].firstLevelPTE = (PTE) 0;
        }
        this->store = store;

        /*
         * Initialize the spaces fault handler table.
         */
        for( i = 0; i < MAXHANDLERS; i++ ) {
                this->faultHandler[i] = 0;
        }
}

/*
 * Space destructor. Boy does this need work!
 */
Space::~Space()
{
        CPUPrintf( "Space::~Space: this=%x\n", this );
        Assert( ! this->lock.held() );
        lock.acquire();              // This could be trouble if 2 processors
                                     // try to delete the space at the same time.
        /*
         * Destroy all of this spaces fault handlers.
         * WHAT IF THERE SHARED. I GUESS THEY NEED TO BE REFERENCE COUNTED.
         */
        for( int i = 0; i < MAXHANDLERS; i++ ) {
                FaultHandler * f = faultHandler[i];
                if( f != 0 )
                        delete f;
        }

        /*
         * Return any second level page tables allocated.
         */
        for( i = 0; i < 256; i++ ) {
                if( this->table[i].secondLevelPTE != 0 ) {
                        //WHAT ABOUT THE VIRTUAL SPACE IT OCCUPIES
                        CPUPrintf( "~Space: THIS REALLY DOESNT WORK\n" );
                        void * paddr = pageToAddr(
                         this->table[i].firstLevelPTE.pageNumber() );
                        (this->store)->deallocate( (char *)paddr, 1 );
                }
        }
}

/*
 * See if an address falls within the range managed by a space.
 */
int
Space::isIn( void * addr )
{
        Debug( "Space::isIn( %x ): this = %x\n", addr, this );
        Assert( this != 0 );

        if( ( addr >= this->baseAddress ) && ( addr <=
           (void *) ( (char *) this->baseAddress + this->length - 1 ) ) )
                return( 1 );
        return( 0 );
}

/*
 * See if an address managed by a Space is currently resident in physical
 * memory (valid).
 */
int
Space::isValid( void * addr )
{
        /*
         * Entry assertions and debugging.
         */
        Debug( "Space::isValid( %x ): this = %x\n", addr, this );
        Assert( this != 0 );
        Assert( this->isIn( addr ) );

        /*
```

```
         * See if the space mappings think the page is valid.
         */
        int l1Index = ((VA) addr).firstLevelIndex();
        int l2Index = ((VA) addr).secondLevelIndex();

        if( this->table[l1Index].secondLevelPTE[l2Index].valid() )
                return( 1 );
        else
                return( 0 );
}

/*
 * Return the first address managed by a Space.
 */
void *
Space::startAddress()
{
        return( this->baseAddress );
}

/*
 * Return the last address managed by a Space.
 */
void *
Space::endAddress()
{
        return( (void *) ( (char *) this->baseAddress + this->length - 1 ) );
}


/*
 * Allocate "count" random pages from the space of type "type".
 * Use "handler" to manage them.
 */
void *
Space::allocate( unsigned int count, FaultHandler * handler,
                allocationType type )
{
        Debug( "Space::allocate( count:%d handler:%x type:%d ): this=%x\n",
                count, handler, type, this );
        if( count == 0 )
                return( 0 );
        Assert( handler != 0 );
        Assert( ( type == prefetch ) || ( type == faultIn ) );

        Assert( ! this->lock.heldByMe() );
        this->lock.acquire();

        /*
         * Figure out the range of pages to allocate.
         */
        Debug( "Space::allocate: vTopPage=%x\n", vTopPage );
        unsigned int start = this->vTopPage;
        void * address = pageToAddr( start );

        Assert( this->isIn( address ) );
        Assert( this->isIn( pageToAddr( start + count - 1 ) ) );

        this->vTopPage += count;

        /*
         * Build the pointer table mappings for the newly allocated pages.
         */
        this->buildMappings( start, count, handler );

        this->lock.release();

        /*
         * Do the prefetching of the pages if requested by the caller.
         */
        if( type == prefetch ) {
                Debug( "Space::allocate: doing prefetch\n" );
                char * addr = (char *) address;
                for( int i = 0; i < count; i++ ) {
                        handler->fixFault( this, addr );
                        addr += PAGESIZE;
                }
        }

        Debug( "Space::allocate: returning %x\n", address );
        return( address );
}

/*
 * Allocate "count" specific pages starting at "base" from the space of type
 * "type". Use "handler" to manage them.
 */
void *
Space::allocate( void * base, unsigned int count, FaultHandler * handler,
                 allocationType type )
{
        /*
         * Entry debugging and assertions.
         */
        Debug( "Space::alloc( base:%x count:%d handler:%x type:%d ): this=%x\n",
                base, count, handler, type, this );
        if( count == 0 )
                return( 0 );
        Assert( handler != 0 );
        Assert( ( type == prefetch ) || ( type == faultIn ) );
        Assert( this->isIn( base ) );
        Assert( this->isIn( (char *) base + (int) pageToAddr( count ) ) );

        Assert( ! this->lock.heldByMe() );
        this->lock.acquire();

        /*
         * Figure out the range of pages to allocate.
         */
        Assert( ( (int)base & 0x1ff ) == 0 );
        unsigned int start = addrToPage( base );
        unsigned int end = start + count - 1;
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        void * address = pageToAddr( start );

        Assert( this->isIn( pageToAddr( start ) ) );
        Assert( this->isIn( pageToAddr( end ) ) );

        /*
         * FIX THIS: it is only a stop-gap solution.
         */
        if ( this->vTopPage <= end )
                this->vTopPage = end + 1;

        /*
         * Build the pointer table mappings for the newly allocated pages.
         */
        this->buildMappings( start, count, handler );

        this->lock.release();

        /*
         * Do the prefetching of the pages if requested by the caller.
         */
        if( type == prefetch ) {
                Debug( "Space::allocate: doing prefetch\n" );
                char * addr = (char *) address;
                for( int i = 0; i < count; i++ ) {
                        handler->fixFault( this, addr );
                        addr += PAGESIZE;
                }
        }

        Debug( "Space::allocate: returning %x\n", address );
        return( address );
}

/*
 * Lookup a fault handler in the per-Space index table.
 * Find an available fault handler index and install the handler
 * if it is not already in the table.
 */
int
Space::convertFaultHandlerToIndex( FaultHandler * handler )
{
        Assert( handler != 0 );

        int freeSlot = -1;

        /*
         * Leave slot 0 empty so that pages without a fault handler
         * return the proper value from the handler() method.
         */
        for( int index = 1; index < MAXHANDLERS; index++ ) {
                if( faultHandler[ index ] == handler )
                        return( index );
                if( ( faultHandler[ index ] == 0 ) && ( freeSlot == -1 ) )
                        freeSlot = index;
        if( freeSlot == -1 )
                return( 0 );

        Assert( ( freeSlot >= 1 ) && ( freeSlot < MAXHANDLERS ) );
        faultHandler[ freeSlot ] = handler;
        return( freeSlot );
}

/*
 * Build the pointer table mappings for "count" pages starting at "start"
 * managed by "handler".
 */
void
Space::buildMappings( unsigned int start, unsigned int count
                      FaultHandler * handler )
{
        /*
         * Entry debuggings and assertions.
         */
        Debug( "Space::buildMappings( start:%x count:%x handler:%x )\n",
                start, count, handler );
        Assert( this->isIn( pageToAddr( start ) ) );
        Assert( this->isIn( pageToAddr( start + count - 1 ) ) );
        Assert( handler != 0 );

        /*
         * Find an available fault handler index and install the handler
         */
        int index = this->convertFaultHandlerToIndex( handler );
        Debug( "Space::buildMappings: Using handler index %x\n", index );

        /*
         * Loop through each page and initialize the pointer table entry for
         * each  Allocate new pointer tables as needed.
         */
        unsigned int page = start;
        for ( unsigned int i = 0; i < count; i++ ) {

                unsigned int l1Index = (page >> 7) & 0xff;
                unsigned int l2Index = page & 0x7f;

                /*
                 * Check if a pointer table needs to be allocated
                 * for this page and get one if it does.
                 */
                if( this->table[l1Index].secondLevelPTE == 0 ) {
                        Debug( "Space::buildMappings: allocating a table\n" );
                        Assert( !(this->table[l1Index].firstLevelPTE.valid()) );

                        PTE * pt = this->allocatePointerTable( page );
                }
                Assert( this->table[l1Index].firstLevelPTE.valid() );
                Assert( this->table[l1Index].secondLevelPTE != 0 );
                Assert( !this->table[l1Index].secondLevelPTE[l2Index].valid() );
```

```
                /*
                 * Set the fault handler index in the pointer table entry.
                 */
                this->table[l1Index].secondLevelPTE[l2Index].handle( index, 3 );
                //HOW DO I GET THE RIGHT PROTECTIONS HERE

                page++;
        }
}

/*
 * Allocate a pointer table for a normal space. The table is allocated out
 * of the CPU's heap space.
 */
PTE *
Space::allocatePointerTable( int page )
{
        Debug( "Space::allocatePointerTable( page:%d )\n", page );
        Assert( this->isIn( pageToAddr( page ) ) );

        unsigned int l1Index = (page >> 7) & 0xff;

        void * pPTaddr = (this->store)->allocate( 1 );
        Debug( "Space::allocatePointerTable: pPTaddr=%x\n", pPTaddr );
        Assert( pPTaddr != 0 );

        Space * heap = Me->heapSpace();
        Debug( "Space::allocatePointerTable: heap = %x\n", heap );
        Assert( heap != 0 );

        void * vPTaddr = heap->allocate( 1 );
        Debug( "Space::allocatePointerTable: vPTaddr=%x\n", vPTaddr );
        Assert( vPTaddr != 0 );
        heap->map( vPTaddr, pPTaddr, 1 );

        Assert( !(this->table[l1Index].firstLevelPTE.valid()) );
        this->table[l1Index].secondLevelPTE = (PTE *) vPTaddr;
        (this->table[l1Index].firstLevelPTE).map( addrToPage( pPTaddr ), 3 );
        //HOW DO WE CHOOSE THE RIGHT PROTECTION LEVEL HERE-

        /*
         * Initialize the new pointer table.
         */
        InitSecondLevelPageTable( (PTE *) vPTaddr );

        return( (PTE *) vPTaddr );
}

void
Space::getPointerTables( unsigned int startPage, unsigned int endPage )
{
        Debug( "Space::getPointerTables( %x, %x )\n", startPage, endPage );
        Assert( NOTREACHED );   // this routine will die soon.
}
```

```
/*
 * Map a virtual page to a physical page frame
 */
void
Space::map( void * page, void * frame )
{
        Debug( "Space::map( page:%x frame:%x ): this = %x\n",
                page, frame, this );
        Assert( this->isIn( page ) );

        Assert( ! this->lock.heldByMe() );
        this->lock.acquire();

        /*
         * Map the virtual page to the physical page frame.
         */
        unsigned int l1Index = ((VA) page).firstLevelIndex();
        unsigned int l2Index = ((VA) page).secondLevelIndex();
        Debug( "Space::map: l1Index=%d l2Index=%d\n", l1Index, l2Index );

        Assert( this->table[l1Index].firstLevelPTE.valid() );
        Assert( this->table[l1Index].secondLevelPTE != 0 );
        Assert( !this->table[l1Index].secondLevelPTE[l2Index].valid() );

        this->table[l1Index].secondLevelPTE[l2Index].map( addrToPage(page), 3 );
        //HOW DO I PUT THE PROPER PROTECTIONS HERE!!!
        Assert( this->table[l1Index].secondLevelPTE[l2Index].valid() );

        this->lock.release();
}

/*
 * handler - return the fault handler function (if any) for a given
 *         new virtual address.
 */
FaultHandler *
Space::handler( void * vaddr )
{
        Debug( "Space::handler( addr:%x ): this=%x\n", vaddr, this );
        Assert( this->isIn( vaddr ) );

        Assert( ! this->lock.heldByMe() );
        this->lock.acquire();

        unsigned l1ix = ((VA) vaddr).firstLevelIndex();
        unsigned l2ix = ((VA) vaddr).secondLevelIndex();
        Debug( "Space::handler: l1ix=%d l2ix=%d\n", l1ix, l2ix );

        if( this->table[l1ix].secondLevelPTE == 0 ) {
                Debug( "Space::handler: invalid L1pte\n" );
                Assert( ! this->table[l1ix].firstLevelPTE.valid() );
                this->lock.release();
                return( 0 );
        }

        PTE * l2pte = &(this->table[l1ix].secondLevelPTE[l2ix]);
```

```
        FaultHandler * theHandler = faultHandler[ l2pte->handlerIndex() ];
        Debug( "Space::handler(%x): index: %x, handler: %x\n",
                vaddr, l2pte->handlerIndex(), theHandler );
        lock.release();
        return( theHandler );
}

/*
 * Constructor for a kernel (heap) space. It does things differently since
 * there is no heap space for it to get things from (since it is one).
 * Kernel spaces get their state and pointer tables from statically allocated
 * Germ virtual memory. They get physical memory like other spaces.
 */
KernelSpace::KernelSpace( Store * store, void * base, int length ) :
        ( store, base, length ) // arguments to parent constructor
{
        Debug( "KernelSpace::KernelSpace( store:%x base:%x length:%x )\n",
                store, base, length );
        Assert( store != 0 );

        /*
         * Get physical pages for this Kernel Space.
         */
        unsigned pages = PageCeiling( sizeof( KernelSpace ) ) >> PAGESHIFT;
        void * pthis = store->allocate( pages );
        Debug( "\tKernelSpace::ctor: pthis = %x (%d pages)\n", pthis,pages );
        Assert( pthis != 0 );

        /*
         * Get virtual pages for this Kernel Space.
         */
        KernelSpace * KernelSpaceAllocator();
        KernelSpace * vthis = KernelSpaceAllocator();

        /*
         * Map into Germ.
         */
        extern void GermMap( unsigned, unsigned, unsigned );
        GermMap( (unsigned) vthis >> PAGESHIFT,
                 (unsigned) pthis >> PAGESHIFT,
                 pages );

        /*
         * OK to set this. Side-effect is to call the parent class (Space)
         * constructor.
         */
        this = vthis;
}

KernelSpace::~KernelSpace()
{
        extern void Halt();

        CPUPrintf( "Kernel destructor called!\n" );
        Halt();
//      GermKernelSpaceDeAllocator( this );
```

```
        this = 0;
}

/*
 * Get pointer tables from pre-allocated Germ virtual memory, not from the
 * heap. Physical memory is allocated from the Space's state Store just like
 * regular spaces.
 */
PTE *
KernelSpace::allocatePointerTable( int page )
{
        Debug( "KernelSpace::allocatePointerTable( page:%d )\n", page );
        Assert( this->isIn( pageToAddr( page ) ) );

        unsigned int l1Index = (page >> 7) & 0xff;

        void * pPTaddr = (this->store)->allocate( 1 );
        Debug( "KernelSpace::getPT: pPTaddr = %x\n", pPTaddr );
        Assert( pPTaddr != 0 );
        extern void * KernelPointerTableAllocate( KernelSpace *, unsigned int );
        void * vPTaddr = KernelPointerTableAllocate( this, page );
        Debug( "KernelSpace::getPT: vPTaddr = %x\n", vPTaddr );
        Assert( vPTaddr != 0 );
        extern void GermMap( unsigned, unsigned, unsigned );
        GermMap( (unsigned) vPTaddr >> PAGESHIFT,
                (unsigned) pPTaddr >> PAGESHIFT, 1 );

        Assert( !(this->table[l1Index].firstLevelPTE.valid()) );
        this->table[l1Index].secondLevelPTE = (PTE *) vPTaddr;
        (this->table[l1Index].firstLevelPTE).map( addrToPage( pPTaddr ), 3 );
        //HOW DO WE CHOOSE THE RIGHT PROTECTION LEVEL HERE-

        /*
         * Initialize the new pointer table.
         */
        InitSecondLevelPageTable( (PTE *) vPTaddr );

        return( (PTE *) vPTaddr );
}

void
KernelSpace::getPointerTables( unsigned int startPage, unsigned int endPage )
{
        Debug( "KernelSpace::getPointerTables(%x,%x)\n",
                startPage, endPage );

        for ( unsigned int page = startPage; page <= endPage; page++ ) {

                unsigned l1Index = (page >> 7) & 0xff;
                unsigned l2Index = page & 0x7f;

                /*
                 * Check if a pointer table needs to be allocated
                 * for this page.
                 */
                if( this->table[l1Index].secondLevelPTE == 0 ) {
```

```
                        Debug( "KernelSpace::getPTs: getting a pte\n" );
                        Assert( !(this->table[l1Index].firstLevelPTE.valid()) );

                        void * pPTaddr = (this->store)->allocate( 1 );
                        Debug( "KernelSpace::getPT: pPTaddr = %x\n", pPTaddr );
                        Assert( pPTaddr != 0 );
                        extern void * KernelPointerTableAllocate( KernelSpace *, unsigned );
                        void * vPTaddr = KernelPointerTableAllocate( this, page );
                        Debug( "KernelSpace::getPT: vPTaddr = %x\n", vPTaddr );
                        Assert( vPTaddr != 0 );
                        extern void GermMap( unsigned, unsigned, unsigned );
                        GermMap( (unsigned) vPTaddr >> PAGESHIFT,
                                (unsigned) pPTaddr >> PAGESHIFT,
                                1 );

                        this->table[l1Index].secondLevelPTE = (PTE *) vPTaddr;
                        (this->table[l1Index].firstLevelPTE).map(
                                addrToPage( pPTaddr ), 3 );
                        //HOW DO WE CHOOSE THE RIGHT PROTECTION LEVEL HERE-

                        /*
                         * Initialize the new second level page table
                         */
                        InitSecondLevelPageTable( (PTE *) vPTaddr );
                }
                Assert( this->table[l1Index].firstLevelPTE.valid() );
                Assert( this->table[l1Index].secondLevelPTE != 0 );
                Assert( !this->table[l1Index].secondLevelPTE[l2Index].valid() );
        }
}

void
Space::map( void * vbase, void * pbase, unsigned int count )
{
        Assert( ! this->lock.heldByMe() );
        this->lock.acquire();

        /*
         * Map the virtual pages to the physical pages.
         */
        unsigned vtop = (unsigned) vbase + ( count * PAGESIZE );
        unsigned page = addrToPage( pbase );
        Debug( "Space::map(%x,%x,%d): vtop=%x, page=%x\n", vbase, pbase,
                count, vtop, page );
        for ( unsigned vaddr = (unsigned) vbase;
                vaddr < vtop; vaddr += PAGESIZE, page++ ) {

                /*
                 * Determine the first and second level page table entries.
                 */
                unsigned l1Index = ((VA) vaddr).firstLevelIndex();
                unsigned l2Index = ((VA) vaddr).secondLevelIndex();

                Assert( this->table[l1Index].firstLevelPTE.valid() );
                Assert( this->table[l1Index].secondLevelPTE != 0 );
                Assert( !this->table[l1Index].secondLevelPTE[l2Index].valid
                /*
                 * Map the physical page.
                 */
                this->table[l1Index].secondLevelPTE[l2Index].map( page, 3 );
                //HOW DO I PUT THE PROPER PROTECTIONS HERE???
                Assert( this->table[l1Index].secondLevelPTE[l2Index].valid() );
        }

        this->lock.release();
}

void *
Space::allocate( unsigned int count )
{
        Assert( ! this->lock.heldByMe() );
        this->lock.acquire();
        Debug( "KernelSpace::allocate( %d ): this=%x\n", count, this );

        /*
         * Figure out the range of pages to allocate.
         */
        Debug( "Space::allocate: vTopPage=%x\n", vTopPage );
        unsigned int start = this->vTopPage;
        void * address = pageToAddr( start );

        Assert( this->isIn( pageToAddr( start + count - 1 ) ) );
        vTopPage += count;

        Debug( "Space::allocate: gettingPointerTable\n" );
        this->getPointerTables( start, vTopPage - 1 );

        this->lock.release();
        Debug( "Space::allocate: returning %x\n", address );
        return( address );
}
```

Owner	russo at m.cs.uiuc.edu
Name	Universe.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:35:29 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 2
Pages printed: 2

Paper size (width, height):
  2560, 3328
Document length:
  4344 bytes
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
May 28 10:03 1987  Universe.c Page 1


/*
 * Universe.c: Universe class implementation. Defines the Spaces accessible
 *              by a processor at any given time
 *
 *      $Header: Universe.c.v 11.0 87/05/21 15:57:22 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Universe.c,v $
 * Revision 11.0  87/05/21  15:57:22  russo
 * Console input and private stores.
 *
 * Revision 10.11  87/05/06  19:30:43  russo
 * added loadContextFor method implementation
 *
 * Revision 10.10  87/05/01  10:45:10  russo
 * turned off debugging
 *
 * Revision 10.9  87/04/25  13:08:47  russo
 * cleaned log
 *
 * Revision 10.5  87/04/22  20:37:50  russo
 * added spaceContaining method.
 *
 * Revision 10.0  87/04/22  07:43:39  russo
 * New Spaces, Universes and CPU objects work. Finally!
 *
 * Revision 9.1  87/04/11  19:50:07  russo
 * initial revision.
 */

#include "Assert.h"
#include "Debug.h"
#include "md_tuneable.h"
#include "Store.h"
#include "Space.h"
#include "CPU.h"
#include "Universe.h"

/*
 * Universe constructor. Sets up the mappings discribed above.
 */
Universe::Universe( Universe * where, PTE * pageTable )
{
        /*
         * Entry debugging and assertions.
         */
        Debug( "Universe::Universe( %x, %x )\n", where, pageTable );
        Assert( this == 0 );
        Assert( where != 0 );
        Assert( pageTable != 0 );

        this = where;
        this->firstLevelPageTable = pageTable;
        this->kernelSpace = 0;
```

```
May 28 10:03 1987  Universe.c Page 2

        this->userSpace = 0;
}

Universe::~Universe()
{
        CPUPrintf( "Universe::~Universe: this = %x\n", this );
        Halt();
}

/*
 * Add a Space to the Universe of addressable spaces on this CPU.
 * This overlays any spaces that are already mapped into the same range
 * of addresses that the new space occupys.
 */
void
Universe::addSpace( Space * aSpace )
{
        Debug( "Universe::addSpace( %x ): this = %x\n", aSpace, this );
        Assert( aSpace != 0 );
        Assert( this != 0 );

        /*
         * Hack to figure out which space is being added. This will go away
         * once the Universe REALLY keeps a list of the mapped in spaces.
         */
        if( aSpace->startAddress() < (void *) TASKLOWADDR )
                this->kernelSpace = aSpace;
        else
                this->userSpace = aSpace;

        /*
         * Copy the first level page table entries in the Space object into
         * the CPUs *real* first level page table. This should be cleaned
         * up to only copy and flush things different than what is already
         * there.
         */
        unsigned int lowFrame = addrToFrame( aSpace->startAddress() );
        unsigned int highFrame = addrToFrame( aSpace->endAddress() );
        Debug( "Universe::addSpace: low/high frame: %d/%d\n",
                lowFrame, highFrame );
        for( int i = lowFrame; i <= highFrame; i++ )
                this->firstLevelPageTable[i] = aSpace->table[i].firstLevelPTE;

        /*
         * VERY inefficient way to flush MMU cache, but for now it works fine.
         */
        WritePTB0( ReadPTB0() );
        WritePTB1( ReadPTB1() );
        Debug( "Universe::addSpace: flushed MMU\n" );
}

/*
 * Return the Space that an address falls in or 0 if the address is in no
 * space currently in the universe.
 */
Space *
```

```
Universe::spaceContaining( void * address )
{
        if( ( address >= this->kernelSpace->startAddress() ) &&
            ( address <= this->kernelSpace->endAddress() ) )
                return( this->kernelSpace );

        else if( ( address >= this->userSpace->startAddress() ) &&
                 ( address <= this->userSpace->endAddress() ) )
                return( this->userSpace );

        else
                return( 0 );
}

void
Universe::loadContextFor( Thread * newThread )
{
        Debug( "Universe::loadContextFor( %x )\n", newThread );
        Assert( this != 0 );
        Assert( newThread != 0 );

        /*
         * Map in the threads VM context. This involves adding any spaces
         * the thread requires to the Universe.
         *
         * This should be fixed to only add whats not already there, or
         * the Universe should be made smart enought to do this.
         *
         * Also we need to fix this to remove what this thread doesnt have
         * access to.
         *
         * Currently this whole mess is hacked for a single thread.
         */
        Space * space = newThread->spaces();
        Debug( "Universe::loadContextFor: space = %x\n", space );

        if( space != 0 )
                this->addSpace( space );
}
```

ORIGINAL PAGE IS
OF POOR QUALITY

Owner	russo at m.cs.uiuc.edu
Name	Thread.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:12:54 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 3
Pages printed: 3

Paper size (width, height):
  2560, 3328
Document length:
  7570 bytes
```

```
/*
 * Thread.c: Implements a generic thread of execution to build
 * higher level kernel processes on top of.
 *
 * Some fields (type, priority) are opaque and just set and read by the
 * kernel. The Space is used to allocate and free memory for the
 * thread.
 *
 *      $Header: Thread.c,v 11.3 87/05/24 05:50:18 russo Exp $
 *      $Locker:  $
 */
/*
 * Revision History:
 *      $Log:   Thread.c,v $
 * Revision 11.3  87/05/24  05:50:18  russo
 * all but important debugging off
 *
 * Revision 11.1  87/05/21  23:34:25  russo
 * working on destructor some
 *
 * Revision 11.0  87/05/21  15:43:04  russo
 * Console input and private stores.
 *
 * Revision 10.22  87/05/12  10:00:18  russo
 * added GermThread class constructor and destructors
 *
 * Revision 10.15  87/05/10  21:09:32  russo
 * altered to accomidate each thread keeping its own interrupt stack.
 * this makes context switching much easier and much more efficient.
 *
 * Revision 10.12  87/05/01  15:47:43  russo
 *
 * Revision 10.0  87/04/22  07:24:53  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  14:55:00  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:22:34  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 1.1  87/02/23  18:11:18  russo
 * Initial revision
 */

#include "Assert.h"
#include "Debug.h"
#include "md_tuneable.h"
#include "md_constants.h"
#include "Thread.h"
#include "Frame.h"
#include "Space.h"
#include "Universe.h"
#include "CPU.h"

/*
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
 * Which .h file should these go in!
 */
#define CPUPSR_I 0x0800
#define CPUPSR_P 0x0400
#define CPUPSR_S 0x0200
#define CPUPSR_U 0x0100

#define CPUPSR_N 0x0080
#define CPUPSR_Z 0x0040
#define CPUPSR_F 0x0020
#define CPUPSR_L 0x0004
#define CPUPSR_T 0x0002
#define CPUPSR_C 0x0001

typedef void (* APFV)();

/*
 * Create a new thread, initialize all the internal fields, and pre-push
 * its initial context onto its interrupt stack.
 */
Thread::Thread( APFV startAddress, int * initialStackPointer, int argument,
        int priority, void * kernelInfo )
{
        /*
         * Enrty Assertions and Debugging.
         */
        Debug( "Thread::Thread(%x, %x) this = %x\n", startAddress,
                initialStackPointer, this );

        Assert( this != 0 );
        Assert( (unsigned) startAddress <= LASTADDRESSABLELOCATION );
        Assert( (unsigned) initialStackPointer <= LASTADDRESSABLELOCATION );

        /*
         * Allocate an initial frame on the threads interrupt stack.
         */
        char * isp = &this->interruptStack[stackSize] - sizeof( struct Frame );
        this->setInterruptStackPointer( isp );
        Debug( "Thread::Thread: interruptStackPointer= %x\n",
                this->interruptStackPointer() );

        struct Frame * context = (struct Frame *) this->interruptStackPointer();
        Debug( "Thread::Thread: initialContext at %x\n", context );

        /*
         * Load the PC, PSR, MOD, SB, SP and FP register copies with their
         * initial contents.
         */
        context->vectorNumber = 0;      // yuck!
        context->pc = (unsigned int) startAddress;
        context->psr = (unsigned short) ( CPUPSR_I|CPUPSR_S|CPUPSR_U );
        context->mod = 0;       // fixing this could solve the lowmem problem
                                // also, it should be loaded with a value
                                // that points to something sensible
        context->sp = (unsigned int) initialStackPointer;
        context->fp = (unsigned int) initialStackPointer;
        /*
         * Save the initial user stack pointer for the destructor to use.
         */
        this->initialUSP = (char *) initialStackPointer

        /*
         * Pass an argument to the new thread in r0. Zero out all the other
         * general purpose registers.
         */
        context->r0 = argument;
        context->r1 = 0;
        context->r2 = 0;
        context->r3 = 0;
        context->r4 = 0;
        context->r5 = 0;
        context->r6 = 0;
        context->r7 = 0;

        /*
         * Fill in the thread object fields that the germ keeps for
         * the kernels use.
         */
        this->priority = priority;
        this->kernelInfo = kernelInfo.
        this->next = 0;
        this->last = 0;
        this->lspaces = 0;
}

/*
 * Free up all the resources owned by a thread and then destory it.
 */
Thread::~Thread()
{
        /*
         * Should deallocate stacks and other resources here
         * and what if its on a queue somewhere?
         */
        CPUPrintf( "Thread::~Thread: this = %x\n", this ).
        Assert( this != 0 );
        Assert( this != Me->currentThread() );
        char * stackPage = (char *) PageFloor( this->initialUSP );
        CPUPrintf( "Thread::~Thread: stackPage = %x\n", stackPage );
        Space * stackSpace = Me->universe()->spaceContaining( stackPage );
        CPUPrintf( "Thread::~Thread: stackSpace = %x\n", stackSpace );
        Assert( stackSpace != 0 );
        //stackSpace->deallocate( stackPage );
        Assert( NOTREACHED );
}

/*
 * Dump the internal contents of a thread
 */
void
Thread::dump()
```

```
{
        struct Frame * f = (struct Frame *) this->interruptStackPointer();

        CPUPrintf( "dump: (this=%x) frame=%x PC=%x PSR=%x MOD=%x SP=%x FP=%x\n",
                this, f, f->pc, f->psr, f->mod, f->sp, f->fp );
        CPUPrintf( "dump: r0=%x r1=%x r2=%x r3=%x r4=%x r5=%x r6=%x r7=%x\n",
                f->r0, f->r1, f->r2, f->r3, f->r4, f->r5, f->r6, f->r7 );
}

int
Thread::isPreemptable()
{
        return( 1 );
}

/*
 * Threads which run with kernel privledges.
 */
KernelThread::KernelThread( APFV startAddress, int * initialStackPointer,
                int argument, int priority, void * kernelInfo )
        : ( startAddress, initialStackPointer, argument,
                priority, kernelInfo )
{
        Debug( "KernelThread::KernelThread() this = %x\n", this);
        Assert( this != 0 );
        struct Frame * f = (struct Frame * ) this->interruptStackPointer();
        Debug( "KernelThread::KernelThread: isp = %x frame = %x\n",
                this->interruptStackPointer(), f );
        f->psr &= ~(CPUPSR_U);
}

KernelThread::~KernelThread()
{
        CPUPrintf( "KernelThread::~KernelThread() this = %x\n", this );
        Assert( NOTREACHED );
}

/*
 * THE germ threads constructor. Later, try to make sure this is only called
 * once or all hell might break loose.
 */
GermThread::GermThread( GermThread * where, APFV startAddress,
                int * initialStackPointer, int argument, int priority,
                void * kernelInfo )
        : ( startAddress, initialStackPointer, argument,
                priority, kernelInfo )
{
        Assert( where != 0 );
        this = where;
        Debug( "GermThread::GermThread() this = %x\n", this );
        struct Frame * f = (struct Frame * ) this->interruptStackPointer();
        Debug( "GermThread::GermThread: isp = %x frame = %x\n",
                this->interruptStackPointer(), f );
        f->psr &= ~(CPUPSR_U | CPUPSR_I);       // system mode, interrupts off
}
```

```
GermThread::~GermThread()
{
        CPUPrintf( "GermThread::~GermThread() this = %x\n", this );
        Assert( NOTREACHED );
}

/*
 * Threads which run with interrupts disabled.
 */
InterruptThread::InterruptThread( APFV startAddress, int * initialStackPointer,
                int argument, int priority, void * kernelInfo )
        : ( startAddress, initialStackPointer, argument,
                priority, kernelInfo )
{
        Debug( "InterruptThread::InterruptThread() this = %x\n", this );
        Assert( this != 0 );
        struct Frame * f = (struct Frame * ) this->interruptStackPointer();
        Debug( "InterruptThread::InterruptThread: frame = %x\n", f );
        f->psr &= ~(CPUPSR_I);
}

InterruptThread::~InterruptThread()
{
        CPUPrintf( "InterruptThread::~InterruptThread() this = %x\n", this );
        Assert( NOTREACHED );
}

int
InterruptThread::isPreemptable()
{
        return( 0 );
}
```

Owner	russo at m.cs.uiuc.edu
Name	TimeSliceThread.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:29:59 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 1
Pages printed: 1

Paper size (width, height):
  2560, 3328
Document length:
  1703 bytes
```

```
/*
 * TimeSliceThread.c:
 *      Set up the timer, wait for it to tick, then set it up again... forever.
 *
 *      $Header: TimeSliceThread.c,v 11.0 87/05/21 15:55:02 russo Exp $
 *      $Locker:  $
 */
/*
 * Revision History:
 *      $Log:   TimeSliceThread.c,v $
 * Revision 11.0  87/05/21  15:55:02  russo
 * Console input and private stores.
 *
 * Revision 10.1  87/05/13  20:29:07  russo
 * initial revision. Split off from KernelMain().
 */

#include "Debug.h"
#include "Assert.h"
#include "Exception.h"
#include "CPU.h"
#include "Vectors.h"
#include "Timer.h"

/*
 * The time slice interrupt Thread code.
 */
void
Ticker( int arg )
{
        Debug( "Ticker( %x )\n", arg );

        InterruptException * clockTick = new InterruptException();
        Me->setException( TIMESLICE_Vector, clockTick );

        Timer * timer = new Timer();
        Debug( "Ticker: timer = %x\n", timer );
        Assert( timer != 0 );

        /*
         * Get our timer initialized and start it running
         */
        while( 1 ) {
                timer->start( 10000, 16 ); // Interrupt 16 in 10 seconds.
                                   // Aren't these nice constants that
                                   // will come back to haunt us some day.
                clockTick->await();

                /*
                 * Restart the timer and acknowledge the interrupt.
                 */
                extern void InterruptAcknowledge();

                timer->stop();
                Debug( "Ticker: Acking interrupt\n" );
                InterruptAcknowledge();
```

```
                Debug( "Ticker: timer restarted and interrupt acknowledged\n" );
        }
}
```

Owner	russo at m.cs.uiuc.edu
Name	Task.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:23:46 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 2
Pages printed: 2

Paper size (width, height):
  2560, 3328
Document length:
  4953 bytes
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * Task.c: task class implementation.
 *
 *      $Header: Task.c,v 11.1 87/05/24 17:01:55 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Task.c,v $
 * Revision 11.1  87/05/24  17:01:55  russo
 * fixed to use new Space allocate methods.
 *
 * Revision 11.0  87/05/21  15:54:58  russo
 * Console input and private stores.
 *
 * Revision 10.8  87/05/12  17:27:12  russo
 * added intiialThread method.
 * And DONT have the constructor add the initial thread to the scheduler.
 *
 * Revision 10.2  87/04/22  15:41:21  russo
 * hack to add tasks space to list of spaces neccessary for a new thead to run.
 *
 * Revision 10.0  87/04/22  07:38:29  russo
 * New Spaces, Universes and CPU objects work. Finally!
 *
 * Revision 9.1  87/04/05  17:20:48  russo
 * fixed so only one fault handler is allocated for all the threads stacks
 * rather than one per stack.
 *
 * Revision 9.0  87/04/04  15:12:41  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:33:54  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:49:41  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 4.1  87/03/08  16:44:24  russo
 * Initial Revision
 */

#include "Debug.h"
#include "md_tuneable.h"
#include "Assert.h"
#include "Space.h"
#include "FaultHandler.h"
#include "Thread.h"
#include "Task.h"
#include "Scheduler.h"

const int ThreadStackSize = PAGESIZE;   // how should this REALLY be decided.

typedef void (* APFV)();

Task::Task( Space * space, APFV initialEntryPoint )
```

```
{
        /*
         * Initial Assertions and Debugging
         */
        Debug( "Task::Task(%x,%x): this=%x\n", space, initialEntryPoint, this );
        Assert( this != 0 );
        Assert( space != 0 );
        Assert( (unsigned int) initialEntryPoint >= TASKLOWADDR );
        Assert( (unsigned int) initialEntryPoint < TASKHIGHADDR );

        /*
         * Set the space member variables.
         */
        this->space = space;

        /*
         * Build the fault handler the threads will initially use to fault
         * their stacks in with. (Make sure this is done before any
         * threads are created in this task.)
         */
        this->stackFaultHandler = new DemandZeroFaultHandler();
        Debug( "Task::Task: this->stackFaultHandler = %x\n",
                this->stackFaultHandler );
        Assert( this->stackFaultHandler != 0 );

        /*
         * Start the initial thread at the initial entry point.
         */
        Debug( "Task::Task: calling startThread\n" );
        this->threads = 0;
        (void) Task::startThread( initialEntryPoint, 0 );
        Debug( "Task::Task: this->threads = %x\n", this->threads );
        Assert( this->threads != 0 );
        Assert( this->threads->next == 0 );
}

/*
 * Task destructor.
 * Delete all the threads in the task then the task space.
 */
Task::~Task()
{
        Printf( "Task::~Task: this = %x\n", this );
        while ( this->threads != 0 ) {
                Thread * t = this->threads;
                this->threads = t->next;
                delete t;
        }
        delete this->space;
}

/*
 * Return a pointer to the Tasks initial thread.
 */
Thread *
Task::initialThread()
```

```
{
        Assert( this != 0 );
        return( this->threads );
}

/*
 * Start up a thread running at the entry point specified in the argument.
 */
Thread *
Task::startThread( APFV entryPoint, int argument )
{
        Debug( "Task::startThread( %x, %x )\n", entryPoint, argument );
        Assert( this != 0 );
        Assert( (unsigned int) entryPoint >= TASKLOWADDR );
        Assert( (unsigned int) entryPoint < TASKHIGHADDR );

        /*
         * Allocate a fill on demand with zeros stack for the thread.
         */
        int stackSize = ThreadStackSize;                // size in bytes
        Assert( (stackSize % PAGESIZE) == 0 );
        int stackPages = stackSize >> PAGESHIFT;        // size in pages
        Assert( stackPages != 0 );

        /*
         * Allocate the space for the stack and set the handler for it.
         */
        int * stack = (int *) space->allocate( stackPages,
                this->stackFaultHandler, faultIn );
        Debug( "Task::startThread: stack(%d bytes) at:%x\n", stackSize, stack );
        Assert( stack != 0 );

        /*
         * Create the thread itself.
         */
        Debug( "Task::startThread: creating the thread\n" );
        Thread * t = new Thread( entryPoint, stack + (stackSize/4), argument,
                0, this );
        Debug( "Task::startThread: thread = %x\n", t );
        Assert( t != 0 );

        /*
         * Add the tasks space to the list of spaces needed to run.
         */
        t->setSpaces( this->space );

        /*
         * Keep the initial thread at the head of the list.
         */
        this->lock.acquire();
        if ( this->threads == 0 ) {
                this->threads = t;
        }
        else {
                Thread * head = this->threads;
                t->next = head->next;
```

```
                head->next = t;
        }
        this->lock.release();

        Debug( "Task::startThread: returning %x\n", t );
        return( t );
}
```

ORIGINAL PAGE IS
OF POOR QUALITY

Owner	russo at m.cs.uiuc.edu
Name	Switch.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:18:20 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 2
Pages printed: 2

Paper size (width, height):
  2560, 3328
Document length:
  4026 bytes
```

```
/*
 * Switch.c: context switching routines
 *
 *      $Header: Switch.c,v 11.3 87/05/27 06:52:38 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Switch.c,v $
 * Revision 11.3  87/05/27  06:52:38  russo
 * dont turn on debugging unless you mean it
 *
 * Revision 11.0  87/05/21  15:43:00  russo
 * Console input and private stores.
 *
 * Revision 10.30  87/05/10  21:09:27  russo
 * altered to accomidate each thread keeping its own interrupt stack.
 * this makes context switching much easier and much more efficient.
 *
 * Revision 10.23  87/05/06  19:43:09  russo
 * use new Universe->loadContextFor method. I'm thinking of including all of
 * this as methods of the CPU object.
 *
 * Revision 10.20  87/05/04  12:44:28  russo
 * fixed problem with disabling interrupts during a switch.
 *
 * Revision 10.3  87/04/22  16:00:48  russo
 * added from list of thread spaces. This removed all knowledge of Tasks
 * from the context switching.
 *
 * Revision 10.0  87/04/22  07:24:49  russo
 * New Spaces. Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  14:54:56  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.1  87/04/04  05:10:39  russo
 * initial revision
 */

#include "Assert.h"
#include "Debug.h"
#include "CPU.h"
#include "Thread.h"
#include "Space.h"
#include "Universe.h"

/*
 * Where, when, and if to disable interrupts here is really leaving me with
 * a sick feeling. --Vince
 */

void
SwitchTo( Thread * newThread )
{
        Debug( "SwitchTo( newThread.%x )\n", newThread );
```

```
        if( newThread == 0 ) newThread = Me->idleThread();
        Assert( newThread != 0 );

        /*
         * Get the current Thread.
         */
        Thread * currentThread = Me->currentThread();

#ifdef DEBUG
        if( newThread == Me->idleThread() )
                CPUPrintf( "SwitchTo: from:%x to:IDLE\n", currentThread );
        else if( currentThread == Me->idleThread() )
                CPUPrintf( "SwitchTo: from:IDLE to:%x\n", newThread );
        else
                CPUPrintf( "SwitchTo: from:%x to:%x\n", currentThread, newThread );
#endif DEBUG

        Assert( currentThread != 0 );
        Assert( currentThread != newThread );

        /*
         * _saveContext() saves the current context so that when the thread is
         * restarted it will appear as if _saveContext() returned 0.
         * The first time it is called it returns the new interruptStackPointer
         * to dispatch the thread with. Another side effect is that when it
         * first returns, interrupts are disabled.
         */
        extern char * _saveContext();
        char * isp;

        Debug( "Calling _saveContext()\n" );
        if( ( isp = _saveContext() ) == 0 ) {
                Debug( "SwitchTo: thread %x, restarted\n", currentThread );
                Assert( currentThread == Me->currentThread() );
                return;
        }
        Debug( "_saveContext returned\n" );

        /*
         * Stuff pushed on the stack from here on will be lost upon restart
         */
        extern void Dispatch( Thread * );

        currentThread->setInterruptStackPointer( isp );
        Dispatch( newThread );
        Assert( NOTREACHED );
}

        /* ARE INTERRUPTS OK ?? */

/*
 * Dispatch a new thread never to return. The context of the thread to dispatch
 * is assumed to be on its interrupt stack.
 */
void
Dispatch( Thread * newThread )
```

```
{
        /*
         * Entry Assertions and Debugging.
         */
        Assert( newThread != 0 );
        Debug( "Dispatch( %x ): interrupt stack will be = %x\n",
                newThread, newThread->interruptStackPointer() );

        /*
         * Remember who were dispatching, load its VM context, and dispatch it.
         */
        extern void _dispatch( char * );

#ifdef DEBUG
        newThread->dump();
#endif
        Me->setCurrentThread( newThread );
        Me->universe()->loadContextFor( newThread );
        Debug( "Dispatch:_dispatch(%x,\n", newThread->interruptStackPointer() );
        _dispatch( newThread->interruptStackPointer() );
        Assert( NOTREACHED
}
```

ORIGINAL PAGE IS
OF POOR QUALITY

Owner	russo at m.cs.uiuc.edu
Name	cswitch.s
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:15:37 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 2
Pages printed: 2

Paper size (width, height):
  2560, 3328
Document length:
  4170 bytes
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * cswitch.s. assembly language context switching routines.
 *
 *      $Header: cswitch.s,v 11.0 87/05/21 15:43:13 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   cswitch.s,v $
 * Revision 11.0  87/05/21  15:43:13  russo
 * Console input and private stores
 *
 * Revision 10.13  87/05/11  08:26:38  russo
 * was saving the wrong stack pointer for restarted threads in _saveContext.
 * I forgot to increment it so as to "pop" off the return address
 *
 * Revision 10.11  87/05/10  21:09:36  russo
 * altered to accomidate each thread keeping its own interrupt stack.
 * this makes context switching much easier and much more efficient.
 *
 * Revision 10.0  87/04/22  07:24:59  russo
 * New Spaces, Universes and CPU objects work. Finally!
 *
 * Revision 9.0  87/04/04  14:55:06  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:13:07  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:43:06  russo
 * Fault handlers hierarchy works, so does the interprocessors vectored
 * interrupt stuff.
 *
 * Revision 1.1  87/02/23  17:56:58  russo
 * Initial revision
 */

/*
 * int _saveContext();
 *
 * Save the context of the current thread and arrange for it to be restarted.
 * It is saved by pushing the current context onto its interrupt stack.
 * It is saved in such a way that when another Thread switches back to it,
 * it will appear as is the call to this procedure simply returned "0".
 * When it is really "called" it returns the interruptStackPointer for the
 * thread to be dispatched with. Another side effect is that it returns with
 * interrupts disabled.
 *
 * We assume nothing but the return address is pushed by a C procedure
 * call since our compiler passes the first two args in r0 and r1
 * We also assume that r0 and r1 are volitile registers across
 * procedure calls.
 */
        .globl  __saveContext
__saveContext:
```

```
        /*
         * Switch stacks to this Threads interrupt stack.
         */
        sprd    sp,r0           /* copy stack pointer before switching */
        sprw    psr,r1          /* save psr before changing it */

        bicpsrw $(0x200+0x800)  /* system stack with no interrupts */

        movw    r1,tos          /* push PSR */
        sprw    mod,tos
        movd    0(r0),tos       /* PC to restart at (on top of user stack) */
        movqd   $0,tos          /* vector number should have some value */

        movqd   $0,tos          /* R0 when restarted */
        movqd   $0,tos          /* R1 when restarted */
        movd    r2,tos
        movd    r3,tos
        movd    r4,tos
        movd    r5,tos
        movd    r6,tos
        movd    r7,tos
        sprd    fp,tos
        addqd   $0x4,r0         /* "pop" return address for restarted thread */
        movd    r0,tos          /* user stack pointer */

        sprd    sp,r0           /* return value = system stack pointer */

        bispsrw $(0x200)        /* switch back to user stack */
        ret     $0              /* the "real" return */


/*
 * void _dispatch( char * interruptStackPointer );
 *
 * Transfer control to another Thread by loading the machine registers
 * from its saved values on its interrupt stack and then "returning" to it.
 * The interrupt stack pointer is activated to do all the work.
 * This can only be called by a thread in kernel mode already.
 *
 * Register Usage:
 *      r0: pointer to the top of the interrupt stack of the Thread to switch to
 */
        .text
        .globl  __dispatch
__dispatch:

        bicpsrw $(0x200+0x800)  /* switch to system stack and make sure */
                                /* interrupts are disabled (because */
                                /* I'm paranoid) */
        lprd    sp,r0

        /*
         * this could be a little cleaner if its too slow. The big problem
         * is that the FIRST time this is called, the interrupt stack is
         * already active.
         */
        movd    tos,r0                          # restore user stack pointer
        bispsrw $0x200
        lprd    sp,r0
        bicpsrw $0x200

        lprd    fp,tos                          # restore frame pointer

        restore [r0,r1,r2,r3,r4,r5,r6,r7]       # restore general registers

        adjspb  $-4             /* pop vector number */
        rett    $0              /* "return" to the new thread */
```

ORIGINAL PAGE IS
OF POOR QUALITY

Owner	russo at m.cs.uiuc.edu
Name	Exception.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:21:02 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 2
Pages printed: 2

Paper size (width, height):
  2560, 3328
Document length:
  3259 bytes
```

```
/*
 * Exception.c: exception class implementations.
 *
 *      $Header: Exception.c,v 11.0 87/05/21 15:42:45 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   Exception.c,v $
 * Revision 11.0  87/05/21  15:42:45  russo
 * Console input and private stores.
 *
 * Revision 10.31  87/05/12  09:31:41  russo
 * use Me->scheduler() rather than runQ
 *
 * Revision 10.27  87/05/10  21:08:49  russo
 * altered to accomidate each thread keeping its own interrupt stack.
 * this makes context switching much easier and much more efficient.
 *
 * Revision 10.22  87/05/01  11:00:13  russo
 * renamed from Event.c
 */

#include "Assert.h"
#include "Debug.h"
#include "Thread.h"
#include "CPU.h"
#include "Exception.h"
#include "Frame.h"
#include "Scheduler.h"

void
Exception::post( struct Frame * frame )
{
        CPUPrintf( "Exception::post( %x )\n", frame );
        Assert( NOTREACHED );
}

SystemException::SystemException( HandlerFunction theHandler )
{
        Assert( theHandler != 0 );
        this->handler = theHandler;
}

SystemException::~SystemException()
{
        Debug( "SystemException::~SystemException()\n" );
        Assert( NOTREACHED );
}

void
SystemException::post( struct Frame * frame )
{
        Debug( "SystemException::post( %x )\n", frame );
        Assert( this->handler != 0 );
        (* this->handler )( frame );
```

```

        Debug( "SystemException::post: returning\n" );
}

InterruptException::InterruptException()
{
        this->awaiter = 0;
}

InterruptException::~InterruptException()
{
}

// Probably should have locks in a lot of the stuff here !!!

void
InterruptException::post( struct Frame * frame )
{
        Debug( "InterruptException::post( %x ) this = %x\n", frame, this );
        Assert( this != 0 );
        Assert( frame != 0 );
        Assert( Me->currentThread() != 0 );

        /*
         * If no-one is awaiting the event, return to who was interrupted.
         */
        if( this->awaiter == 0 ) {
                CPUPrintf( "Un-awaited InterruptException\n" );
                return;
        }

        /*
         * Arrange for the current thread to be restarted where it was
         * interrupted.
         */
        Me->currentThread()->setInterruptStackPointer( (char *) frame );

        /*
         * We are not real happy with this, probably the idle thread
         * SHOULD be enqued, but with the lowest possible priority.
         */
        if( Me->currentThread() != Me->idleThread() ) {
                Assert( Me->scheduler() != 0 );
                Me->scheduler()->add( Me->currentThread() );
        }

        /*
         * Dispatch the thread awaiting the event.
         */
        extern void Dispatch( Thread * );

        Thread * t = this->awaiter;
        this->awaiter = 0;
        Dispatch( t );
        Assert( NOTREACHED );
}
```

```
void
InterruptException::await()
{
        Debug( "InterruptException::await(): Thread %x awaiting %x\n",
                Me->currentThread(), this );
        Assert( this != 0 );
        Assert( this->awaiter == 0 );
        Assert( Me->currentThread() != 0 );

        this->awaiter = Me->currentThread();

        extern void SwitchTo( Thread * );

        if( Me->scheduler() == 0 )
                SwitchTo( 0 );
        else
                SwitchTo( Me->scheduler()->removeNext() );
}
```

ORIGINAL PAGE IS
OF POOR QUALITY.

Owner	russo at m.cs.uiuc.edu
Name	FaultHandlers.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:21:56 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 2
Pages printed: 2

Paper size (width, height):
  2560, 3328
Document length:
  7021 bytes
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * FaultHandlers.c
 *
 *      $Header: FaultHandlers.c,v 11.5 87/05/24 23:13:21 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification history:
 *      $Log:   FaultHandlers.c,v $
 * Revision 11.5  87/05/24  23:13:21  russo
 * use two argument space::map
 *
 * Revision 11.4  87/05/24  16:45:38  russo
 * cleaning up
 *
 * Revision 11.1  87/05/24  05:08:20  russo
 * in the middle of re-doing fault handlers.
 *
 * Revision 11.0  87/05/21  15:54:39  russo
 * Console input and private stores.
 *
 * Revision 10.2  87/04/22  20:44:12  russo
 * get space containing the faulting address from the Universe.
 *
 * Revision 10.0  87/04/22  07:38:13  russo
 * New Spaces, Universes and CPU objects work. Finally!
 *
 * Revision 9.0  87/04/04  15:12:23  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:33:40  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:49:30  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 1.1  87/03/19  17:36:12  johnston
 * Initial revision
 */

#include "Debug.h"
#include "Assert.h"
#include "VM.h"
#include "Store.h"
#include "Space.h"
#include "CPU.h"
#include "mi_tuneable.h"
#include "md_tuneable.h"

/*
 * Common code for all subclasses of FaultHandler.
 */
void
FaultHandler::fixFault( Space * space, void * address ) {
        /*
         * Eventually this should do the equivalent of the UNIX SIGSEGV
```

```
         * and terminate the Task.
         */
        Printf( "FaultHandler::fixFault( space:%x address:%x): this=%x\n",
                space, address, this );
        Assert( NOTREACHED );
}

/*
 * StoreManager class.
 */
StoreManager::StoreManager( Store * store )
{
        Debug( "StoreManager::StoreManager( store:%x ): this = %x\n",
                store, this );
        Assert( this != 0 );
        Assert( store != 0 );
        this->storeBeingManaged = store;
}

StoreManager::~StoreManager()
{
        Debug( "StoreManager::~StoreManager(): this = %x\n", this );
        Assert( NOTREACHED );
}

void
StoreManager::fixFault( Space * space, void * address )
{
        Debug( "StoreManager::fixFault( space:%x address:%x ): this = %x\n",
                space, address, this );
        Assert( space != 0 );
        Assert( space->isIn( address ) );
        Assert( this != 0 );
        Assert( this->storeBeingManaged != 0 );

        void * frame = this->storeBeingManaged->allocate( 1 );
        void * page = (void *) PageFloor( address );
        Debug( "StoreManager::fixFault: page=%x, frame=%x\n", page, frame );
        space->map( page, frame );
}

/*
 * Common code for all subclasses of DemandFillFaultHandler (allocateAndMap).
 *
 * Demand fill fault handlers are those that, upon a fault, allocate a page
 * from the store, map it into the VM Space, and the fill it with something.
 * What they fill it with depends on the particular sub-class of
 * DemandFillFaultHandler being referenced.
 */
void
DemandFillFaultHandler::allocateAndMap( Space * space, void * address )
{
        extern Store * MainStore;

        Debug( "DemandFillFH::allocateAndMap( space:%x address:%x):\n",
                address, space );
        Assert( space != 0 );

        /*
         * Determine the virtual page number that faulted.
         */
        void * virtualBase = (void *) PageFloor( (unsigned) address );

        /*
         * Allocate a physical page from the store and map it in.
         */
        Debug( "DemandFillFaultHandler::allocateAndMap: getting a page\n" );
        void * physicalBase = MainStore->allocate( 1 );
        Assert( physicalBase != 0 );
        space->map( virtualBase, physicalBase, 1 );
        Debug( "DemandFillFaultHandler::allocateAndMap: virt:%x phys:%x\n",
                virtualBase, physicalBase );
}

/*
 * Constructor for the demand filler class fault handler class that uses
 * the filler class to fill the pages with. Its only job is to set the filler
 * member equal to the argument. The filler class member function fillPage()
 * will be used by this classes fixFault() member function to fill the
 * faulting page after it has been allocated and mapped in.
 */
DemandFillerClassFaultHandler::
DemandFillerClassFaultHandler( Filler * filler ) {
        /*
         * Allocate space for state information.
         * WE SHOULD CHECK FOR DUPLICATES TO SAVE SPACE AND ALSO DO REFERENCE
         * COUNTING SINCE WE REALLY DONT KNOW WHEN TO DELETE ONE
         */
        Assert( this != 0 );
        Debug( "DemandFillerClassFaultHandler::DemandFiller( %x) this=%x\n",
                this, filler );
        this->filler = filler;
}

/*
 * Destructor for the above demand fill fault handler sub-class.
 */
DemandFillerClassFaultHandler::~DemandFillerClassFaultHandler() {
        Printf( "DemandFillFaultHandler destructor called''\n" );
        delete filler;
}

/*
 * Demand fill (using a Filler) fault handler subclass fixFault() routine.
 * Allocate a page from the store, map it in to the Tasks VM, and the
 * call the filler members fillPage() member funtion to initialize the page.
 */
void
DemandFillerClassFaultHandler::fixFault( Space * space, void * address )
{
        Debug( "DemandFillerClassFH::fixFault(space:%x address%x): this=%x\n",
                space, address, this );
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        allocateAndMap( space, address );
        /*
         * Call the fillers fillPage member function to fill in the page.
         */
        Debug( "DemandFillerClassFaultHandler::fixFault: calling filler\n" );
        filler->fillPage( address );
        Debug( "DemandFillerClassFaultHandler::fixFault: returning\n" );
}

/*
 * The constructore for the subclass of demand fill fault handler that fills
 * the faulting page with zeros.
 */
DemandZeroFaultHandler::DemandZeroFaultHandler()
{
        /*
         * Allocate space for state information.
         * WE SHOULD CHECK FOR DUPLICATES TO SAVE SPACE AND ALSO DO REFERENCE
         * COUNTING SINCE WE REALLY DONT KNOW WHEN TO DELETE ONE
         */
        Assert( this != 0 );
        Debug( "DemandZeroFaultHandler::DemandZeroFaultHandler: this = %x\n",
                this );
}

/*
 * The destructor for the above.
 */
DemandZeroFaultHandler::~DemandZeroFaultHandler()
{
        Printf( "DemandZeroFaultHandler destructor called''\n" );
}

/*
 * The fixFault routine for DemandZeroFaultHandler. This calls the parent
 * classes allocateAndMap() member function to get a page and make it
 * addressable. Then it simply fills the new page with zeros.
 */
void
DemandZeroFaultHandler::fixFault( Space * space, void * address )
{
        Debug( "DemandZeroFH::fixFault( space:%x address%x ): this=%x\n",
                space, address, this );
        allocateAndMap( space, address );

        /*
         * Calculate the base address of the page and fill it with zeros.
         */
        extern void ClearMemory( void *, unsigned );

        void * virtualBase = (void *) PageFloor( (unsigned) address );
        Debug( "DemandZeroFaultHandler::fixFault: ClearMemory( %x, %x )\n",
                virtualBase, PAGESIZE );
        ClearMemory( virtualBase, PAGESIZE );
        Debug( "DemandZeroFaultHandler::fixFault: returning\n" );
}
```

Owner	russo at m.cs.uiuc.edu
Name	COFFRoutines.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:25:37 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 3
Pages printed: 3

Paper size (width, height):
  2560, 3328
Document length:
  9941 bytes
```

```
/*
 * COFFRoutines.c: routines to deal with setting up a space to be loaded
 *      from a UNIX COFF (Common Object File Format) image.
 *
 *      $Header: COFFRoutines.c,v 11.4 87/05/27 05:34:18 russo Exp $
 *      $Locker:  $
 */
/*
 * Revision History:
 *      $Log:   COFFRoutines.c,v $
 * Revision 11.4  87/05/27  05:34:18  russo
 * debug off
 *
 * Revision 11.3  87/05/25  05:41:21  russo
 * debugging on
 *
 * Revision 11.2  87/05/24  17:04:38  russo
 * spelling error
 *
 * Revision 11.1  87/05/24  16:44:34  russo
 * switched to new Space allocate routines.
 *
 * Revision 11.0  87/05/21  15:54:27  russo
 * Console input and private stores.
 *
 * Revision 10.0  87/04/22  07:38:08  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  15:12:17  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:33:35  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 7.0  87/03/25  12:49:05  russo
 * Fault handler hierarchy works, so does interprocessor vectored interrupts.
 *
 * Revision 6.1  87/03/22  17:55:46  russo
 * Initial Revision.
 */

#include "Debug.h"
#include "Filler.h"
#include "FaultHandler.h"
#include "File.h"
#include "Space.h"
#include "md_tuneable.h"
#include "/usr/include/a.out.h"

void bcopy( char *, char *, int );

typedef void (* APFV)();

/*
 * Setup a Space to be demand filled from a COFF image in a File.
 */
```

```
APFV
SetupSpaceFromCOFFImage( Space * space, File * file )
{
        /*
         * THIS IS A LOT OF STUFF TO PUT ON THE STACK, IS IT TOO MUCH??
         */
        union {
                struct filehdr filehdr;
                struct aouthdr aouthdr;
        } u;
        struct scnhdr scnhdr;
        int filePointer = 0;

        Debug( "SetupSpaceFromCOFFImage(%x, %x)\n", space, file );
        Assert( space != 0 );
        Assert( file != 0 );

        /*
         * Read the file header and check if it's ok (magic = NS32GMAGIC).
         */
        Debug( "SetupSpaceFromCOFFImage: Reading file header\n" );
        file->readRecords(filePointer, (char *) &u.filehdr,
                sizeof( struct filehdr ) );
        filePointer += sizeof( struct filehdr );
        if( u.filehdr.f_magic != NS32GMAGIC ) {
                Printf( "SetupSpaceFromCOFFImage: Bad File Magic Number %x\n",
                        u.filehdr.f_magic );
                return( (APFV) 0 );
        }

        /*
         * Read the a.out header from the file, then read the .text, .data,
         * and .bss section headers and remember the useful bits of
         * information in each.
         */
        Debug( "SetupSpaceFromCOFFImage: Reading a.out header\n" );
        file->readRecords( filePointer, (char * ) &u.aouthdr,
                sizeof( struct aouthdr ) );
        filePointer += sizeof( struct aouthdr );

        Debug( "SetupSpaceFromCOFFImage: Reading .text section header\n" );
        file->readRecords( filePointer, (char *) &scnhdr,
                sizeof( struct scnhdr ) );
        filePointer += sizeof( struct scnhdr );
        long textScnPtr = scnhdr.s_scnptr;

        Debug( "SetupSpaceFromCOFFImage: Reading .data section header\n" );
        file->readRecords( filePointer, (char *) &scnhdr,
                sizeof( struct scnhdr ) );
        filePointer += sizeof( struct scnhdr );
        long dataScnPtr = scnhdr.s_scnptr;

        Debug( "SetupSpaceFromCOFFImage: Reading .bss section header\n" );
        file->readRecords( filePointer, (char *) &scnhdr,
                sizeof( struct scnhdr ) );
        filePointer += sizeof( struct scnhdr );
        /*
         * Calculate the addresses and sizes of each of the sections
         * The data and bss sections are assumed to be contiguous and are
         * are treated as one section.
         */
        APFV entryPoint = (APFV) u.aouthdr.entry.
        void * textStart = (void *) u.aouthdr.text_start;
        void * dataStart = (void *) u.aouthdr.data_start;

        Debug( "SetupSpaceFromCOFFImage: textStart=%x, dataStart=%x entry=%x\n",
                textStart, dataStart, entryPoint );

        Assert( ((unsigned)textStart % PAGESIZE) == 0 );
        Assert( (unsigned)textStart >= TASKLOWADDR );
        Assert( (unsigned)textStart < TASKHIGHADDR );
        Assert( ((unsigned)dataStart % PAGESIZE) == 0 );
        Assert( (unsigned)dataStart >= TASKLOWADDR );
        Assert( (unsigned)dataStart < TASKHIGHADDR );

        int textSize = u.aouthdr.tsize;
        int textPages = textSize / PAGESIZE;
        if ( ( textSize % PAGESIZE ) != 0 ) {
                textPages++;
        }
        Assert( (unsigned)(textStart + textSize) < TASKHIGHADDR );
        Assert( (char *)entryPoint >= textStart );
        Assert( (char *)entryPoint <= (textStart + textSize) );

        int dataSize = u.aouthdr.dsize + u.aouthdr.bsize;
        int dataPages = dataSize / PAGESIZE;
        if ( ( dataSize % PAGESIZE ) != 0 ) {
                dataPages++;
        }
        Assert( (unsigned)(dataStart + dataSize) < TASKHIGHADDR );
        Debug( "SetupSpaceFromCOFFImage: textSize=%x, textPages=%d\n",
                textSize, textPages );
        Debug( "SetupSpaceFromCOFFImage: dataSize=%x, dataPages=%d\n",
                dataSize, dataPages );

        /*
         * Build a fault handler for the .text section.
         */
        Debug( "SetupSpaceFromCOFFImage: Building .text section filler\n" );
        COFFSectionFiller * textFiller = new COFFSectionFiller( file, textStart,
                textSize, textScnPtr );
        Debug( "textFiller = %x\n", textFiller );
        Assert( textFiller != 0 );
        Debug( "SetupSpaceFromCOFFImage: Building .text fault handler\n" );
        DemandFillerClassFaultHandler * textFaultHandler =
                new DemandFillerClassFaultHandler( textFiller );
        Debug( "textFaultHandler = %x\n", textFaultHandler );
        Assert( textFaultHandler != 0 );

        /*
         * Allocate the pages for the .text section and install the
```

```
         * fault handler created above. Fault handlers should probably
         * be deleted by the Space destructor when it is fully implemented,
         * but they also may need to be deleted when they are no longer
         * needed. For example, if all the data has been faulted in, the
         * fault handler to load it is no longer needed since all of the
         * data pages will have there fault handlers replaced by some form
         * of swapping fault handler if their memory is reclaimed. Granted,
         * if a clean page is chosen for replacement, the original fault
         * handler will still be needed to reclaim it when needed.
         */
        Debug( "SetupSpaceFromCOFFImage: Allocating the text pages\n" );
        char * text = (char *) space->allocate( textStart, textPages,
                textFaultHandler, faultIn );
        Debug( "SetupSpaceFromCOFFImage: text = %x\n", text );
        Assert( text != 0 );

        /*
         * Build the fault handler for the .data and .bss section.
         */
        Debug( "SetupSpaceFromCOFFImage: Building .data and .bss filler\n" );
        COFFSectionFiller * dataFiller = new COFFSectionFiller( file, dataStart,
                dataSize, dataScnPtr );
        Debug( "dataFiller = %x\n", dataFiller );
        Assert( dataFiller != 0 );
        Debug( "SetupSpaceFromCOFFImage: Building .data .bss fault handler\n" );
        DemandFillerClassFaultHandler * dataFaultHandler =
                new DemandFillerClassFaultHandler( dataFiller );
        Debug( "dataFaultHandler = %x\n", dataFaultHandler );
        Assert( dataFaultHandler != 0 );

        /*
         * Allocate the pages for the .data and .bss sections and install the
         * fault handler created above.
         */
        Debug( "SetupSpaceFromCOFFImage: Allocating the data pages\n" );
        char * data = (char *) space->allocate( dataStart, dataPages,
                dataFaultHandler, faultIn );
        Debug( "SetupSpaceFromCOFFImage: data = %x\n", data );
        Assert( data != 0 );

        /*
         * Return the entry point address for this Space. This value is
         * passed to the Task constructor when it is called.
         */
        Debug( "SetupSpaceFromCOFFImage: returning %x\n", entryPoint );
        return( entryPoint );
}

/*
 * Constructor to fill in the internal fields of a COFF section filler.
 */
COFFSectionFiller::COFFSectionFiller( File * file, void * start,
                                      int size, long location )
{
        /*
         * Entry Assertions and Debugging.
```

```
	 */
	Debug( "COFFSectionFiller::COFFSectionFiller( %x, %d, %x ) this = %x\n",
		start, size, location, this );
	Assert( this != 0 );
	Assert( file != 0 );

	/*
	 * initialize the internal fields.
	 */
	Assert( ((unsigned)start % PAGESIZE) == 0 );
	this->file = file;
	this->sectionStart = start;
	this->sectionLength = size;
	this->fileLocation = location;
}

/*
 * Destructor for COFFSection fillers. DONT delete the file here (I think?)
 */
COFFSectionFiller::~COFFSectionFiller()
{
	Printf( "COFFSectionFiller Destructor called''\n" );
}

void
COFFSectionFiller::fillPage( void * faultingAddress )
{
	/*
	 * round faulting address down to its page number.
	 */
	Debug( "COFFSectionFiller::fillPage(%x)\n", faultingAddress );
	void * virtualBase = (void *) PageFloor( (unsigned) faultingAddress );
	Debug( "COFFSectionFiller::fillPage: virtualBase = %x\n", virtualBase );

	/*
	 * copy in the pages data from the COFF section.
	 */
	int whichOffset = (int) virtualBase - (int) sectionStart;
	long source = fileLocation + whichOffset;
/*
 * SHOULD FILE LOCK THE READS AND WRITES HERE??
 * ALSO, WHILE IM THINKING ABOUT IT. WHO DELETES THE FILE??
 * I THINK COFFSpace should be a subclass of task space and its destructor
 * should delte the coff file.
 */
	Debug( "COFFSectionFiller::fillPage: offset %x in section (addr=%x)\n",
		whichOffset, virtualBase );

	if( source > ( fileLocation + sectionLength ) ) {
		extern void ClearMemory( void *, int );
		Debug( "COFFSectionFiller::fillPage: filling with zeros\n" );
		ClearMemory( virtualBase, PAGESIZE );
	}
	else {
		Debug( "COFFSectionFiller::fillPage: copying in page\n" );
		file->readRecords( source, (char *) virtualBase, PAGESIZE );
```

```
	}
	Debug( "COFFSectionFiller::fillPage: returning\n" );
}

/*
 * Copy 'count' bytes from "from" to "to". Replace this with a nicely optimized
 * assembly language version eventually.
 */
void
bcopy( char * from, char * to, int count )
{
	while( count != 0 ) {
		*to++ = *from++;
		count--;
	}
}
```

Owner	russo at m.cs.uiuc.edu
Name	GoOnline.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:12:00 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 2
Pages printed: 2

Paper size (width, height):
  2560, 3328
Document length:
  5607 bytes
```

```
/*
 * GoOnline.c: First C language routine called by each procesor after doing
 *             initialization. Spin in a loop and wait for an inital thread
 *             to be added to the scheduler. Once one is there switch control
 *             to it and pray we NEVER return.
 *
 *      $Header: GoOnline.c,v 11.1 87/05/21 16:57:57 russo Exp $
 *      $Locker:  $
 */
/*
 * Revision History:
 *      $Log:   GoOnline.c,v $
 * Revision 11.1  87/05/21  16:57:57  russo
 * CPU member changed names
 *
 * Revision 11.0  87/05/21  15:42:49  russo
 * Console input and private stores.
 *
 * Revision 10.43  87/05/13  06:35:44  russo
 * took out setException for CONSOLE_Vector
 *
 * Revision 10.41  87/05/12  17:37:12  russo
 * set up to be called by the germs initial thread.
 *
 * Revision 10.36  87/05/02  15:42:54  russo
 * split into a germ half(this) and a kernel half(KernelEntry)
 *
 * Revision 10.23  87/04/30  16:09:21  russo
 * build a private InterruptException for the clock and and intial thread to
 * handle the clock interrupt. Dispatch that thread. It will await, the
 * event mechanism should restart the idle thread. When the interrupt
 * occurs the clock thread should run again, reset the clock, and await the
 * interrupt again.
 *
 * Revision 10.15  87/04/28  10:35:50  russo
 * build a default exception handler and install it for all vectored exceptions.
 * Then re-install the proper ones for those we care about
 *
 * Revision 10.0  87/04/22  07:24:41  russo
 * New Spaces, Universes and CPU objects work, Finally!
 *
 * Revision 9.0  87/04/04  14:54:48  russo
 * Multiple threads and timer interrupts.
 *
 * Revision 8.0  87/03/29  15:22:17  russo
 * _new and _delete added for memory management. Also, class interrupts work.
 *
 * Revision 1.1  87/02/23  18:11:18  russo
 * Initial revision
 */

#include "Assert.h"
#include "Debug.h"
#include "md_tuneable.h"
#include "Thread.h"
#include "Exception.h"
```

```
#include "Lock.h"
#include "Space.h"
#include "Store.h"
#include "Universe.h"
#include "CPU.h"
#include "Vectors.h"

/*
 * Default exception handler until the kernel installs what it wants.
 * There SHOULD be no exceptions until the kernel is running and decides its
 * ready.
 */
extern void Uncaught( struct Frame * );
SystemException DefaultException( Uncaught );

/*
 * A Lock to arbitrate kernel creation, and a pointer to the initial kernel
 * Space. The pointer is set by the processor that creates it and shared by all
 * processors.
 */
static Lock kernelLock;
static KernelSpace * InitialKernelSpace = 0;

/*
 * The C language entry point for the Germ thread.
 */
void
GoOnline( unsigned int ID )
{
        /*
         * Say hello Gracie.
         */
        Assert( Me != 0 );
        Printf( "Processor %x is online (ID=%x)\n", Me->id(), ID );

        /*
         * Sanity checks of the boot code.
         */
        Assert( Me->id() == ID );
        Assert( Me->universe() != 0 );
        Assert( Me->idleThread() != 0 );
        Assert( Me->currentThread() != 0 );
        Assert( Me->currentThread() == Me->idleThread() );
        Assert( Me->scheduler() == 0 );
        Assert( Me->threadToDelete() == 0 );
        Assert( Me->privateStore() != 0 );
        Assert( Me->globalStore() != 0 );

        extern int InterruptsDisabled();
        Assert( InterruptsDisabled() );

        /*
         * Install the initial (default) exception handlers. Again, no
         * exceptions should happen until the kernel is executing.
         */
        Debug( "GoOnline: installing default (panic) exceptions\n" );
        for( int i = 0; i < Me->numberOfVectoredExceptions( ); i++ )
                Me->setException( i, &DefaultException );

        /*
         * Test to see if the kernel has been built yet and, if not, build it.
         * If it has been built, someone else has been here first.
         * In this case just skip the call, release the lock, and continue.
         */
        kernelLock.acquire();
        if( InitialKernelSpace == 0 ) {
                Debug( "GoOnline: building InitialKernelSpace\n" );

                /*
                 * Build the initial Kernel.
                 */
                extern char * VirtualPrivateMemory;

                char * kernelStart = VirtualPrivateMemory + 0x10000;
                InitialKernelSpace = new KernelSpace( Me->globalStore( ),
                        kernelStart, /* MAXKERNELMEM */
                         (char *) TASKLOWADDR - kernelStart );
                Debug( "GoOnline: InitialKernel = %x\n", InitialKernelSpace );
                Assert( InitialKernelSpace != 0 );
        }
        kernelLock.release( );

        /*
         * Add the kernel space, InitialKernelSpace, to this
         * processors Universe. Also set the heapSpace instance
         * variable in the CPU object.
         */
        Debug( "GoOnline: Adding initial kernel %x to universe\n",
                InitialKernelSpace );
        Assert( InitialKernelSpace != 0 );
        Me->universe( )->addSpace( InitialKernelSpace );
        Me->setHeapSpace( InitialKernelSpace );

        /*****************************************************************
         * By the time everybody gets here there is a kernel (heap) space
         * available to allocate things out of and added to the processors
         * Universe. All initial set up should also be completed.
         *
         * All thats left do is to turn the current Tread over to the kernel
         * by calling KernelMain( ), who will install all the exception
         * handlers and do other kernel initilization things, then start
         * dispatching threads.
         *****************************************************************/

        extern void KernelMain( unsigned int );

        Debug( "GoOnline: Calling KernelMain\n" );
        KernelMain( ID );
        Assert( NOTREACHED );
}
```

ORIGINAL PAGE IS
OF POOR QUALITY

Owner	russo at m.cs.uiuc.edu
Name	ExceptionHandlers.c
Account	173
Site	Dept. of Computer Science
Printer	24/300
SpoolDate	Thu May 28 10:30:54 1987

```
JobHeader on
JamResistance On
Language printer
formwidth 132
formsperpage 2
outlines on

IMAGEN Printing System, Version 2.2, Serial #86:2:85
Page images processed: 3
Pages printed: 3

Paper size (width, height):
  2560, 3328
Document length:
  7759 bytes
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
 * ExceptionHandlers.c. Various exception handling routines.
 *
 *      $Header: ExceptionHandlers.c,v 11.11 87/05/27 05:35:13 russo Exp $
 *      $Locker:  $
 */
/*
 * Modification History:
 *      $Log:   ExceptionHandlers.c,v $
 * Revision 11.11  87/05/27  05:35:13  russo
 * debug off
 *
 * Revision 11.10  87/05/26  21:35:25  russo
 * switched the switch to if...then....elseif since the compiler
 * seems to be totally hosed.
 *
 * Revision 11.9  87/05/26  07:09:10  johnston
 * Debugging on.
 *
 * Revision 11.8  87/05/26  06:24:52  johnston
 * Made ABTTrap use PanicPrintf instead of CPUPrintf.
 *
 * Revision 11.7  87/05/24  22:32:25  russo
 * attempt to solve the case statement wierdness.
 *
 * Revision 11.4  87/05/24  06:13:03  russo
 * trying to figure out whats happening with KILLTHREAD
 *
 * Revision 11.2  87/05/24  05:09:52  russo
 * fixed calls to fixFault.
 *
 * Revision 11.1  87/05/21  16:50:45  russo
 * switched ways of deleteing threads
 *
 * Revision 11.0  87/05/21  15:54:34  russo
 * Console input and private stores.
 *
 * Revision 10.40  87/05/17  14:04:43  russo
 * added terminating threads to the per-cpu delete queue.
 *
 * Revision 10.37  87/05/16  12:45:38  russo
 * renamed from boot/TrapCatchers.c
 */

#include "md_tuneable.h"
#include "Debug.h"
#include "Assert.h"
#include "VM.h"
#include "FaultHandler.h"
#include "SVCs.h"
#include "Space.h"
#include "CPU.h"
#include "Thread.h"
#include "Task.h"
#include "Frame.h"
#include "Scheduler.h"
```

```
extern void Halt();

void
DumpFrame( struct Frame * frame )
{
        CPUPrintf( "DumpFrame(%x): PC=%x PSR=%x MOD=%x Vector=%d\n", frame,
                frame->pc, frame->psr, frame->mod, frame->vectorNumber );
        CPUPrintf( " r[0-7]= %x:%x:%x:%x:%x:%x:%x:%x sp=%x fp=%x\n",
                frame->r0, frame->r1, frame->r2, frame->r3, frame->r4,
                frame->r5, frame->r6, frame->r7, frame->sp, frame->fp );
}

void
ABTTrap( struct Frame * frame )
{
        /*
         * Grab the MMU registers we'll need.
         */
        unsigned msr = ReadMSR();
        unsigned eia = ReadEIA();
//#ifdef DEBUG
        PanicPrintf( "Abort Trap: eia=%x msr=%x pc=%x psr=%x\n",
                eia, msr, frame->pc, frame->psr );
        //DumpFrame( frame );
        //((MSR *) &msr)->printf();
//#endif
        /*
         * Get the faulting address from the EIA
         */
        unsigned address = ((EIA) eia).address();

        /*
         * Get the Space containing the faulting address
         * Wierd things will happen if mapping of 'Me' and the Universe
         * is not set up yet. We really shouldn't put ABTTrap in the trap
         * table until were ready.
         */
        Space * faultingSpace =
                Me->universe()->spaceContaining( (void *) address );
        Debug( "ABTTrap: faulting space = %x\n", faultingSpace );
        Assert( faultingSpace != 0 );

        /*
         * Handle the fault. (first, check for conditions we don't understand.)
         */
        Assert( !((MSR) msr).BPTError() );
        Assert( !((MSR) msr).BPR() );
        Assert( !((MSR) msr).BPTReadError() );
        Assert( !((MSR) msr).BPTStatError() );
        Debug( "ABTTrap: Mode of fault: %s.\n",
                ((EIA) eia).txPTB() ? "user" : "supervisor" );

        /*
         * The first thing we do is see if we faulted on an address in a Space
         * that has grown. If this is what happened, then all we have to do
```

```
	 * either add some new entrys to the first level page table in the
	 * Universe or invalidate the cached MMU descriptor.
	 */
	if( faultingSpace->isValid( (void *)address ) ) {
		Debug( "ABTTrap: faulted on a valid address\n" );
		Me->universe()->addSpace( faultingSpace );
			// add space also has the nice feature of flushing
			// the MMU for us and adding any new PTEs.
		Debug( "ABTTrap: retrying\n\n" )
		return;
	}

	/*
	 * If it wasn't already valid, get the fault handler (if there is one)
	 * to fix the page.
	 */
	FaultHandler * theHandler = faultingSpace->handler( (void *) address );
	if( theHandler == 0 ) {
		/*
		 * In the future this should just kill the thread (task?).
		 */
		PanicPrintf( "fault with no handler!\n" );
		Halt();
	}

	/*
	 * Call the Space fault handler and return. This results in
	 * retrying/restarting the instruction which caused the fault.
	 */
	Debug( "Calling handler (*%x)(space:%x address%x)\n",
		theHandler, faultingSpace, address );
	theHandler->fixFault( faultingSpace, (void *) address );
	Debug( "ABTTrap: Handler returned - retrying\n\n" );
}

void Touch( int x ) {}

typedef void (* APFV)();

void
SVCTrap( struct Frame * frame )
/*
 * This is all a bit of a hack until the "into the kernel" object calls work.
 */
{
	Debug( "SVC Trap\n" );
#ifdef DEBUG
	DumpFrame( frame );
#endif

	Debug( "SVC(%d) USP=%x\n", frame->r0, frame->sp )
	char * ap = (char *) (frame->sp + 4);

	int svc = frame->r0;

	if( svc == PRINTF_SVC ) {
```

```
			// should fault if needed (what a *#^!$ hack!)
		Touch( ** (char **)ap );
		CPUPrintf( *(char **)ap );
	}
	else if( svc == KILLTASK_SVC ) {
		CPUPrintf( "KillTask SVC Called\n" );
		Halt();
	}
	else if( svc == KILLTHREAD_SVC ) { // terminate your own execution
		CPUPrintf( "KillThread SVC Called\n" );
		Assert( Me->currentThread() != 0 );
		Assert( Me->currentThread() != Me->idleThread() );
		Assert( Me->threadToDelete() == 0 );

		/*
		 * Set the current processors Thread to delete
		 * to the current Thread.
		 * The idle Thread cleans up and deletes the exiting
		 * Thread, so switch to it.
		 */
		extern void SwitchTo( Thread * );

		Me->setThreadToDelete( Me->currentThread() );
		CPUPrintf( "KillThread set threadToDelete\n" );
		SwitchTo( 0 );		// relinquish the CPU
		Assert( NOTREACHED );
	}
	else if( svc == STARTTHREAD_SVC ) {
		Debug( "STARTTHREAD_SVC at %x\n", *(int *)ap );
		Debug( "currentThread = %x\n", Me->currentThread() );
		Assert( Me->currentThread() != 0 );
		Task * thisTask =
			(Task *) (Me->currentThread()->getKernelInfo());
		Debug( "thisTask = %x\n", thisTask );
		Assert( thisTask != 0 );
		APFV arg1 = *(APFV *) ap;
		int arg2 = *(int *) ( ap + 4 );
		Debug( "call startThread(%x, %x)\n", arg1, arg2 );
		Thread * newThread = thisTask->startThread( arg1, arg2 );
		Debug( "newThread = %x\n", newThread );
		Assert( newThread != 0 );
		Assert( Me->scheduler() != 0 );
		Me->scheduler()->add( newThread );
	}
	else {
		CPUPrintf( "Invalid SVC (%d) Called\n", frame->r0 );
		Halt();
	}
	frame->pc++;
	Debug( "SVC: set new pc to %x\n", frame->pc );
}

void
Uncaught( struct Frame * frame )
{
	CPUPrintf( "Uncaught vectored exception (#%x)\n", frame->vectorNumber );
```

```
        DumpFrame( frame );
        Halt();
}

/*
 * Be sure to call InterruptAcknowledge for strays if we ever do anything but
 * Halt().
 */

/*
 * Catch all exceptions and re-direct them to the proper handlers.
 *
 * This should be an Exception handler in it own right'.
 */
void
ExceptionCatcher( struct Frame * frame )
{
        Debug( "ExceptionCatcher( %x ): vector %x\n", frame,
                frame->vectorNumber );
        Assert( frame->vectorNumber >= 0 );
        Assert( frame->vectorNumber < Me->numberOfVectoredExceptions() );
        Exception * exception = Me->exception( frame->vectorNumber );
        Debug( "ExceptionCatcher: exception = %x\n", exception );
#ifdef ASSERT
        if( exception == 0 ) {
                        DumpFrame( frame );
                        Assert( exception != 0 );
        }
#endif
        exception->post( frame );
                        // post worries about context saveing and restoring if
                        // neccessary or just calling a subroutine otherwise.
                        // It also worries about which stack to use, etc...
        Debug( "ExceptionCatcher: post returned\n" );
}
```

ORIGINAL PAGE IS
OF POOR QUALITY

APPENDIX D

Cross–Architecture Procedure Call

Raymond Brook Essick

THE CROSS-ARCHITECTURE PROCEDURE CALL

BY

RAYMOND BROOKE ESSICK IV

B.S., University of Illinois, 1981
M.S., University of Illinois, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1987

Urbana, Illinois

ACKNOWLEDGEMENTS

Many friends helped me to complete this thesis. I would like to thank all of them for their encouragement, support and friendship. Names that stand out among these friends include Barb Cicone, Dirk Grunwald, Anda Harney, Rob Kolstad, Jim Purtilo, Malcolm Railey, John Shilling, and the members of the Systems Research Group. In particular, Rob helped generate the insight that formed the basis for this thesis. Other friends that helped to generate a pleasant working environment and answered the questions that I couldn't answer include Jeff Donnelly, Kurt Horton, and Paul Richards.

I could not have completed this thesis without the support of my advisor, Professor Roy H. Campbell. He provided invaluable intellectual and (invaluable!) financial support during my research. My financial support came through NASA grant NSG–1471. This grant funds the EOS project, which is directed by Professor Roy Campbell. I would also like to thank the members of my committee: Ralph Johnson, Duncan Lawrie, Jane Liu, and Dan Reed.

My wife, Jeanna, provided more support than I can ever acknowledge. She helped me through the difficult spots and focused my attention on other things when I needed to step back from my work for a while. Without her, the last 5 years would not have been nearly as enjoyable.

TABLE OF CONTENTS

CHAPTER 1.

INTRODUCTION

Workstations provide good interactive computing environments that have consistent user response times and support many devices suitable for interactive work including bit mapped displays, mice, and keyboards. Supercomputers supply large amounts of sequential and vector computing power. However, they do not provide a cost effective interactive environment. This thesis introduces the *Cross–Architecture Procedure Call*, a software architecture that allows applications to exploit both systems. Cross–Architecture Procedure Calls (or *CAPC*s) combine virtual memory, high speed networking, and compatible data representations to accelerate an application's computations without modifying its code. CAPCs allow workstation applications to use, on a demand basis, faster or more expensive processors as compute servers so that each of an applications functions can be executed by the most appropriate processor.

Workstations offer a number of advantages to a centralized timesharing system. A network of workstations is more resistant to complete failure than a single system. (However, the larger number of components increases the probability of partial failure.) Failure of a single workstation usually does not prevent other workstations from functioning. Experimental software, frequent (or unexpected) reboots, and different operating systems are easier to manage with a connected network of workstations. The incremental costs to enhance a network of workstations are small.

Several costs offset these advantages. A system administrator must deal with many workstations instead of a single, central machine. System resources, which used to be centralized and easily managed, are now distributed across many systems. One resource lost in the transition from a centralized system to a network of workstations is the sequential processing power of the large timesharing CPU. Many tasks do not require the high CPU bandwidth available in the centralized system. However, some tasks do require this bandwidth; moving these tasks to workstations causes unacceptable increases in their execution time.

The most common way to reduce or eliminate this increase in execution time is to ship the entire application to a supercomputer. This batch–oriented technique does not exploit the interactive features of the workstation.

Another way to decrease execution time is to restructure sequential applications into concurrent applications and then run them concurrently on many processors. Processor configurations range from tightly coupled systems sharing common memory to loosely coupled systems that communicate over networks such as Ethernet [64]. The tightly coupled systems provide a centralized multiprocessor environment. They do not offer the same set of interactive tools available on a workstation. Some systems automatically restructure sequential applications to execute concurrently on their many processors [11,42]. Other systems do not restructure the application; the user must manually convert sequential applications to concurrent applications.

Loosely–coupled systems can provide large amounts of processing power. At this time, however, network communications times are dramatically slower than local memory references. This communications overhead affects the choice of algorithms. Algorithms that generate less traffic between systems replace simple and fast algorithms that work well in tightly–coupled systems (with low communications costs). These replacement algorithms may increase the processing demands of the application while reducing the network traffic.

Remote Procedure Calls provide a mechanism to execute subroutines on remote loosely–coupled processors [16,66]. Applications can be partitioned so that CPU–intensive routines execute on the supercomputer and other routines execute on the workstation. RPCs have several restrictions that affect how an application is partitioned. RPC client and server processes do not share the same address space. Thus, pointer–based structures do not transfer well to a RPC environment and routines on different processors can not share the same global variables. All communications between routines must be through the argument list. RPC systems require stub routines and special compile–time operations to generate instructions to transfer control between the client and server systems. These factors affect how an application can be partitioned in an RPC environment.

Programs often spend large fractions of their execution time in small sections of their code. This is often paraphrased as the *90–10* rule: programs spend 90% of their time in 10% of the code. Sometimes, performance can be improved by selecting more appropriate or efficient algorithms. In other cases, the algorithm in use is already optimal. In these cases, the only way to make that section of code execute faster is to place it on a faster processor. Often, this 10% of the code is contained within several subroutines. Therefore, these subroutines should be moved to a faster processor.

This thesis proposes a software architecture for executing programs in an environment with workstations and supercomputers. Applications can exploit each processor's particular features. Interactive portions of an application can execute on the workstation. CPU–intensive portions of an application can execute on the supercomputer. This architecture provides a standard process model — an application existing in a single address space. Our architecture usually does not require any restructuring of applications programs.

Our new architecture partitions applications between workstations and supercomputers while meeting the following criteria. These criteria reflect our goals not to require restructuring of applications and to exploit the features of both workstations and supercomputers.

- The user need not restructure or recode his applications.
- The programmer can specify an application's partitioning. Changes to this partitioning do not require changes to the application source code.
- Interactive tasks execute on the workstation. That is, the workstation is not used as a simple terminal to submit jobs to the supercomputer.
- CPU–intensive tasks execute on the supercomputer.
- Optimization techniques, such as vector operation and parallel operations, specific to certain architectures are still useful for code segments executed on those architectures.
- The compilers for each system need not be modified; a modified loader combines the output from the respective compilers into an executable file.
- The operating system resolves issues of control transfer and data transfer between systems.

1.1. CLASP Overview

This thesis proposes the CLASP software architecture. CLASP, an acronym for *Cross–architecture Address SPace*, implements a new foundation for the traditional process model. This new foundation allows heterogeneous CPUs to share the virtual address space of a process. Tasks within the application execute on the CPU most appropriate to their needs — interactive response, large amounts of CPU bandwidth, vector processing. CLASP identifies a level of homogeneity necessary to implement this sharing. It also mitigates dissimilarities between the processor architectures such as register sets and stack frame formats. CLASP makes these differences transparent to the programmer.

CLASP allows heterogeneous CPUs with different performance characteristics and potentially different instruction sets to operate in a single address space. This allows portions of an application to be executed by the most appropriate processor. Where other research efforts have augmented standard addressing schemes to provide *remote addresses*, CLASP makes a single address space accessible to multiple heterogeneous CPUs [78]. A novel aspect of the CLASP architecture is the inclusion of instructions for different processor architectures within the same address space. These instructions are placed in different regions of the address space.

Our architecture introduces a new control transfer mechanism, the *Cross Architecture Procedure Call* (or *CAPC*). Like the Remote Procedure Call (or *RPC*), the Cross Architecture Procedure Call transfers control between processors. RPCs introduce new calling sequences into the application code to transfer control between processors. CAPCs do not modify the subroutine calling sequence in the application code. In CAPCs, both local and remote subroutine calls use the standard subroutine call and return instructions. The CLASP kernel detects calls that refer to remote subroutines, packages arguments, and transfers the control thread to the remote processor. When a subroutine is moved from the local processor to the remote processor, the CAPC system does not require any changes to the source or compiled instances of procedures that invoke the migrated subroutine.

Applications are prepared for this architecture by the new CLASP loader, which links separately compiled routines into a single executable image. This loader recognizes the different object formats for various processor architectures and resolves the cross-architecture references. It provides the operating system kernel with the information necessary to detect control transfers (e.g., procedure calls and returns) that cross architecture boundaries. Routines that execute on specific architectures are compiled for those architectures. Some frequently called routines (e.g., sqrt()) can be replicated. Duplicate copies of these routines, each compiled for a different architecture, are loaded into the executable file. Calls to any of these routines can be directed to the local instance of that routine, saving the network overhead of a remote call. The loader chooses which instance to use when resolving references to these routines.

Trees, lists, and other pointer–based data structures are difficult and sometimes impractical to implement in distributed computing models without a shared address space. The SUN Remote Procedure Call dereferences pointers to pass individual elements of a pointer–based structure [16,17]. Pointer dereferencing is adequate for situations where single structures are passed by pointer instead of value. Others have advocated the use of subroutines to encapsulate access to pointer–based structures [46,66]. This approach implies changing (or deliberately designing) applications to encapsulate accesses to these structures. The CLASP software architecture solves this problem by ensuring that the context for a pointer (i.e., its address space) can be transferred to the remote processor. Applications may use pointers as handles to objects and for true pointer–based structures without concern about where a procedure is implemented.

CLASP uses demand paging to move arguments and data to the server. As an example, binary searches through large sorted arrays can be efficient because the accessed portions of the array are transferred to the remote processor on demand instead of prepaging the entire array to the server. Pages, once transferred to the server, remain on the server until they are required by the client processor. Pages used only by the client remain on the client; pages used only by the server will be transferred to and remain on the server. Pages of data used by both processors migrate between hosts on demand.

Although CLASP appears to be an approach to distributed computing, it is actually an extension of the traditional single–processor model onto a new underlying implementation that provides improved performance. CLASP mimics this single processor model but allows the most appropriate CPU to execute appropriate parts of the problem. It does not require restructuring of applications. Only portions that execute on a remote processor need to be recompiled. The choice of which processor performs a specific routine affects only the processing rate for that procedure. The choice does not alter the semantics for that procedure nor its interactions with other procedures in the address space.

1.2. Thesis Organization

Chapter 2 describes some of the work that motivated this thesis. Chapter 3 provides a formal definition of the components of the CLASP system. Chapter 4 describes our prototype CLASP system and the protocols it uses to communicate between processors. Chapter 5 presents performance figures for our CLASP prototype. It also points out factors to consider when partitioning an application to use CAPCs. The final chapter summarizes our results and considers some additional research based on the CAPC concept.

CHAPTER 2.

RELATED WORK

This chapter presents a summary of other research directed towards sharing resources and providing language level support for this sharing. Two primary areas are explored: language features that provide access to other processors and system designs that support shared resources. We do not discuss language constructs that support concurrent processes. Although concurrency provides a foundation to reduce the execution time of an application, it usually requires that programmers manually restructure source code to use different algorithms. In this thesis, we direct our efforts towards making a single control thread execute faster.

The first portion of this chapter concentrates on language mechanisms. The approaches discussed in these sections represent different mechanisms for transferring a control thread between processors.

The next portion of this chapter discusses network filesystems. Network filesystems remove restrictions on where applications can execute by making the data required for those applications available from almost any processor. This flexibility encourages a migration towards workstations that provide effective work environments.

After the network filesystem discussion, this chapter presents several distributed systems. Two types of distributed systems are discussed. The first class extends the operating system to include many component systems. The second class of distributed operating system moves traditional operating system services, like filesystems, out of the kernel and into application programs.

We discuss multiprocessor systems in the penultimate section of the chapter. Many of these systems are suitable as the compute servers that we want to use for the compute–intensive portions of our applications. Some of these systems use special compilers to convert sequential applications to concurrent applications. This allows applications to use all of the processors in these systems and reduce the computation time.

The chapter closes with a summary of these research efforts. We show how these systems do not meet all of the criteria presented in chapter 1.

2.1. Language–based Partitioning Mechanisms

In this section, we describe several language–based partitioning mechanisms. These mechanisms allow applications to perform computations on remote processors. Each of these techniques imposes some restrictions on the applications program. Some require applications to be recoded in a new language. Others restrict the operations and data types that can be used in remote operations.

The Remote Procedure Call uses the subroutine call abstraction as a logical point to transfer control between processors. The Interface Compilers described improve the implementation of Remote Procedure Call systems.

Distributed Path Pascal provides access to remote objects within the Path Pascal languages. Although the source code must be changed to use remote objects, the changes are minor and do not affect the existing interface to an object. Object–oriented systems provide support for remote operations. Eden, Smalltalk, and other object–oriented systems provide support for objects that reside on other systems. However, these object–oriented systems and Distributed Path Pascal require that an application be coded in the appropriate language to use these features.

2.1.1. The Remote Procedure Call

Remote Procedure Calls — also called RPC — build on the control transfer intrinsic to a procedure call and extend this control transfer across machine boundaries.[1] Nelson argues that RPC is a satisfactory and efficient programming language primitive for constructing distributed systems [66]. The RPC model consists of a *client* process that invokes subroutines implemented by a *server* process. The client and server are separate processes and execute in their own address space. Servers advertise a set of subroutines that clients can invoke.

Because the client and server do not share a common address space, RPC clients and servers can only communicate through the parameter lists and return values of the subroutines advertised by the server.[2] Client and server procedures can not pass information through global variables because the two processors do not share an address space. In his thesis, Nelson suggests that procedural interfaces be used for access to global variables [46,66]. In a general RPC system, a server can call a routine on the client to retrieve a global variable. By encapsulating access to global variables, programs can be partitioned (and repartitioned) across clients and servers at later times with less chance of error in routines

[1] An RPC call does not have to go to another machine. The RPC server can be located on the same CPU, but within a different process.

[2] We discount the possibility that the client and server exchange information through a shared filesystem. This approach suffers from the same limitations: the client and server must take explicit action to transfer information between each other. Although filesystem communications might provide for more information to be passed in a single transaction, it is not transparent.

that depend on access to global variables.

Some RPC implementations require special calling sequences to invoke routines on the server [17,19]. Figures 2.1 and 2.2 depict the client and server code segments used to invoke a remote procedure using the Sun RPC implementation.

RPC implementations often work across heterogeneous hardware. The client and processor may have different instruction sets, processing speeds, and data representations. To accommodate the different data representations, arguments and results of remote procedures are coerced to a standard representation before being sent to the peer process. When received, these values are again coerced, this time from the standard order to the order used by the receiving processor [8,13]. This makes the RPC mechanism available across a diverse combinations of processors. To implement an RPC system, a process must be able to coerce data between its internal representation and the network standard representation.

The generality of a standard network representation introduces several costs to a program's execution. Systems must always convert data to the standard representation before sending it across the network; the recipient must always convert from the standard representation to its internal representation.[3] If the client and server share a data representation that differs from the network standard order, RPC subroutines convert the data twice when it could have been passed without change. A number of processors use the Network Standard Order for their internal representation [4–7,47,48]. These machines can send data across the network without any format conversions. However, RPC calls marshall their parameters or results into a single buffer as part of the conversion procedure. Often, this

[3] The recipient must perform this conversion. Intermediate sites, providing gateway functions, pass the data without conversion.

```
/*
 *      result = foo(5, 7, 'c', "a string");
 */
struct foo_arglist
{
    long    arg1;
    long    arg2;
    char    arg3;
    char   *arg4;
};

caller ()
{
    long    result;
    int     failed;
    struct foo_arglist  fooargs;

    fooargs.arg1 = 5;
    fooargs.arg2 = 7;
    fooargs.arg3 = 'c';
    fooargs.arg4 = "a string";

    failed = callrpc (HOST, PROGRAM, VERSION, PROCEDURENUMBER,
            xdr_fooargs, &fooargs, xdr_long, &result);
    if (failed)
        exit (1);
    /*
     * "result" contains return value from foo.
     */
}

xdr_fooargs (xdrs, fp)
register    XDR *xdrs;
struct fooargs *fp;
{
    return (xdr_long (xdrs, &fp->arg1) && xdr_long (xdrs, &fp->arg2) &&
            xdr_char (xdrs, &fp->arg3) && xdr_string (xdrs, &fp->arg4));
}
```

Figure 2.1
Sample SUN RPC Client Code

```
foohandler (rqstp, transp)
struct svc_req *rqstp;
SVCXPRT * transp;
{
    long    value;
    struct foo_arglist  fooargs;

    if (!svc_getargs (transp, xdr_fooargs, &fooargs))
    {
        fprintf (stderr, "unable to decode arguments\n");
        exit (1);
    }
    value = foo (fooargs.arg1, fooargs.arg2, fooargs.arg3, fooargs.arg4);
    if (!svc_sendreply (transp, xdr_long, &value))
    {
        fprintf (stderr, "cant reply to caller\n");
        exit (1);
    }
    svc_freeargs (transp, xdr_fooargs, &fooargs);
    return;
}

long    foo (arg1, arg2, arg3, arg4)
long    arg1;
long    arg2;
char    arg3;
char   *arg4;
{
    /* foo procedure implemented here */
}

main ()
{
    registerrpc (PROGRAM, VERSION, PROCEDURENUMBER,
            foohandler, xdr_fooargs, xdr_long);
    svc_run ();
    printf ("returned from svc_run -- bad news\n");
    exit (1);
}
```

Figure 2.2
Sample SUN RPC Server Code

copy operation is performed even though no format conversion is done.

Different data types require different conversions between internal and network representations. To provide the correct mappings, the RPC systems must provide type information for procedure arguments. Constructs that abuse type information can fail in an RPC environment. As an example, the C *cast* operation allows programs to interpret the same bit string several different ways.

The RPC client and server have separate address spaces. This separation provides an easy way to replace the code of a remote procedure dynamically. New RPC client requests can be sent to the new implementation of the server. RPC servers can be removed as their clients finish execution.

Because RPC clients and server exist in separate address spaces, clients are not bound to specific servers until runtime. Different instances of the client may interact with different server programs. This can be used to reconfigure programs at runtime with a minimum of effort. In an RPC–based GKS implementation, the client process chooses a server appropriate for the desired output device [76]. The X window package uses the deferred client/server binding in the same manner [43–45].

Some RPC systems do not provide mechanisms for routines on the server to invoke arbitrary functions on the server [2,3,16]. The relationship between the client and server is a strict master/slave relationship.

2.1.2. Interface Compilers

The set of procedures, their arguments, and their results make up a protocol between the RPC client and server.[4] To ensure that both the client and server obey the same protocol, a number of interface compilers have been developed.

Interface compilers accept definitions of a procedure's arguments and results. They output two code segments — one for the client and one for the server. The client code segment is a set of routines that resembles a local instance of the subroutine. This client routine packages its arguments, transmits an RPC request to the server, retrieves the results and unpacks them. The server code segment unpacks an RPC request and invokes the true subroutine, which is implemented on the server.

Interface compilers reduce the complexity of managing RPC interfaces. Courier, *rpcgen*, and related compilers take a specification of the input and output parameters for a procedure and generate the necessary subroutines to package the arguments, transmit them to a server processor, unpack the arguments, execute the routine on the server, and return the results to the client processor.

Interface compilers do not change the semantics of the remote procedure call. They simplify the specification, implementation, and management of the set of routines that an RPC client can invoke on on an RPC server. In the next two sections, we discuss two interface compilers: Xerox Courier and Sun RPCL. Both compilers allow specification of the available procedures and the arguments and results for those procedures. Both compilers generate the data conversion and packaging routines to translate between internal

[4] Protocols often are associated with message passing. See [59] for a discussion of how the Remote Procedure Call and message passing are duals. Message passing can model RPC and RPC can model message passing.

representation and the network standard order. The Courier system also generates local stub routines that can be called using local procedure call mechanisms. These stub routines perform the actual RPC call.

2.1.2.1. Xerox Courier

Xerox's Courier language provides a way to specify procedure interfaces. The Courier compiler translates the simple specification into the appropriate client and server stub routines. Courier-generated stub routines resolve data representation differences between hosts by converting parameters and results to a standard order. Simple data types are converted to a standard ordering for transmission across a network. Complex data types, such as records, are decomposed until they are a collection of simple data types. The pieces of a complex data type are reassembled on the remote host, using that host's alignment and data representation. If the Courier specification is correct, the resulting stub routines will be correct.

Figure 2.3 contains a Courier specification for the subroutine foo. This concise definition can be compared to the more complex notation in figures 2.1 and 2.2 for Sun RPC. With Courier, a client invokes the local stub routine for foo. This stub routine packages the arguments and sends them to the remote processor.

```
Foo: PROGRAM = BEGIN

      -- Foo entry point

      foo : PROCEDURE [arg1 : INTEGER, arg2 : INTEGER,
            arg3 : CARDINAL, arg4 : ARRAY 32 OF CARDINAL ]
            RETURNS [ result : INTEGER ]
      = 0;

END.
```

Figure 2.3
Sample Courier code

Xerox has developed other RPC stub compilers. The Lupine compiler generates RPC stubs for the Cedar language. Versions of the Courier and Lupine compilers generate stubs for languages like Mesa, Interlisp, C, Smalltalk, and others [26].

2.1.2.2. Sun RPCL

Sun has developed an interface compiler for their Remote Procedure Call Language, RPCL [16,19]. RPCL uses a C–like syntax to specify the datatypes, procedures, and versions of an RPC server. The RPCL compiler, *rpcgen*, produces header files for inclusion in applications code, generates the XDR routines for converting data to the external data representation, and produces a server program to register the program. RPCL permits one argument for each remote procedure. RPCL does not build stub routines that allow users to invoke remote routines with multiple arguments like:

foo (1, 2, 'c', "a string");

Instead, the argument must be collected into a single structure to be passed to the server.

Figure 2.4 shows the RPCL specification for the foo subroutine depicted in figures 2.1 through 2.3. RPCL automates the generation of the packaging and conversion routines and structures (the structure `foo_arglist` and the routine *xdr_fooargs*() in figure 2.1). It does not remove the extra work in the *caller*() routine in figure 2.1.

The single argument restriction becomes important when the local or remote routines' calling conventions are not under the control of the programmer configuring them for an RPC environment. In such cases, the programmer can not change the procedure interface so that it accepts a single complex argument. Instead, he must manually generate an extra layer of interface routines to convert between an expanded argument list and a single structure. The Courier RPC language provides these stubs; RPCL does not.

```
struct foo_arglist
{
    long    arg1;
    long    arg2;
    char    arg3;
    string arg4[];
};

program RBEPROGRAM
{
    version RBEVERSION
    {
        long    foohandler (foo_arglist) = 1;
    } = 1;
} = 100000;
```

Figure 2.4
Sample SUN RPCL Code

2.1.3. Network Data Segment

The *network data segment*, or NDS, is an extension of the remote procedure call that allows the client and server to share access to global variables [56]. Like RPC, an NDS task comprises two processes — one on the client system and one on the server system. NDS adds a data segment that both client and server can access. This allows the client and server to share access to global variables. These processes share a specific range of their address space. Each processor can see changes made to data in this address range by the other processor. The two processors do not share physical memory; instead, they use networking hardware — like Ethernet — to transfer pages between themselves. The NDS software architecture is targeted for languages like FORTRAN, where the client and server routines both access variables in COMMON.

The virtual memory subsystem and networking capabilities of the host systems provide the means to share parts of the address space between the processors. The shared data resides at the same addresses in each process' virtual memory address space. Figure 2.5 shows the address space layout for the NDS architecture. The NDS software keeps one copy of each page in this range. These pages are demand paged between the processors as needed. When the client accesses a page, the page migrates to the client's physical memory. When the server makes a reference to that page, it generates a page fault. The pagefault software retrieves the non–resident page from the client processor instead of the local backing store. Shared pages of the address space stay resident on the processor that last accessed them. In long running programs, shared pages eventually will reside on the processor where they are accessed. Pages accessed by both processors will move between the processors as needed.

The NDS software architecture supports languages like FORTRAN, where all variables have static addresses. NDS does not work as well with stack based languages, such as C, where many variables are stored on the stack. The NDS client and server processors each have their own stack; neither processor can access data stored in the other processor's stack. Thus, the NDS architecture does not provide complete support for languages that store variables on the stack.

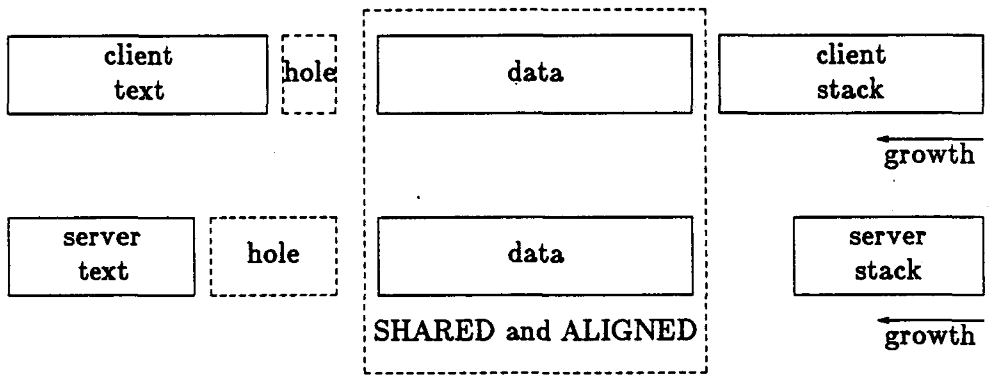

Figure 2.5
NDS Address Space Layout

To execute an NDS program, the user must have two programs — an executable for the client and an executable for the server. The code for each processor is stored in a separate executable file. Special NDS compilers require information about routines are implemented locally and remotely. For remote routines, the NDS compiler generates RPC stubs to invoke the routine on the remote processor. Because the client and server programs are the result of several different compiler runs, the compilers for both processors must guarantee the same ordering, alignment, and length of operands. Client and server compilers must assign locations with COMMON blocks in the same fashion.

2.1.4. Distributed Path Pascal

Distributed Path Pascal combines the parallelism features of a concurrent language with access to remote resources [55]. Kolstad's thesis describes an enhancement to Path Pascal that provides *remote objects.* These remote objects may reside on other machines. References to entry procedures of a remote object are processed in an RPC fashion. Distributed Path Pascal packages the arguments and sends a message to a server on the remote host. The server then unpacks the arguments and invokes the object's entry procedure. When the procedure terminates, control returns to the calling process.

Distributed Path Pascal provides a language–level mechanism for access to remote resources. It does, unfortunately, require changes to the source code to use the remote operations. These changes are restricted to a change in the declaration of the object.

Another shortcoming in the Distributed Path Pascal approach is that all operations on a single object occur on the same processor. If an object encapsulates a large database, all operations on the database occur on the same processor. Distributed Path Pascal does not allow a fast lookup operation to be implemented on the workstation and an expensive re–ordering of the database to be implemented on a compute server.

2.1.5. Object–Oriented Systems

Some object–oriented systems provide mechanisms to access objects on remote hosts. Many of these systems uses messages as a communications mechanism between objects.

Messages to remote objects require extra processing at some level to move the message across a communications link. Remote objects do not change the nature of the language.

These systems require different programming techniques. For example, large FORTRAN codes do not port directly to these systems.

For example, the Eden system provides location–independent names for individual objects in the system [23]. Each Eden object, or *eject*, has its own address space. Objects can move between processors with the same architecture. An eject defining a matrix might define functions to invert the matrix, lookup specific elements, and replace specific elements. All of these operations are implemented on the same architecture.

2.2. Networked Filesystems

Network Filesystems direct their efforts towards sharing data among systems. While these efforts are less ambitious than a complete distributed system, they provide a useful level of sharing — particularly in UNIX–like systems that rely on the filesystem for most communications between processes. Network Filesystems provide a framework that can be extended to build a distributed system [34].

Network filesystems are useful because they allow users to work on arbitrary machines — instead of being forced to work on the machine that holds their data. If a user can access his supercomputer files from both the supercomputer and a workstation, he will often choose the workstation for its superior work environment. With network filesystems (that allow users to access supercomputer files from workstations) and the results of this thesis (which allows users to access supercomputer cycles from workstations), users can exploit both systems easily.

There are many designs for, and implementations of, network filesystems. These include: Xerox Alpine, SUN ND, SUN NFS, and IBM RVD [18,20,26,85]. Distributed systems like LOCUS and the Newcastle Connection provide network filesystems as part of their larger goals.

2.2.1. Xerox Networked Filesystems

Researchers at Xerox PARC have developed several distributed filesystems. These include Juniper, the Interim File Server, and the Alpine filesystem.

The Juniper filesystem, also known as the Xerox Distributed Filesystem, is an effort to support access to shared databases in the Xerox environment — an environment where all shared files are stored on file servers [65]. Because the XDFS filesystem is targeted for shared database systems, it had to provide support for common database operations. XDFS provides random access files and atomic transactions. Juniper was implemented on the Alto, a 16 bit workstation. After experimenting with this implementation, it was discovered that the performance was slow (but tolerable) and that the server frequently crashed. Server recovery took over an hour. In addition, new software systems being developed at Xerox provided a new basis for a more efficient and more robust file server [26,80]. This new filesystem is the Alpine filesystem.

Another file server in use at Xerox is the *IFS* or *Interim File Server.* IFS differs from a true filesystem in that it moves the entire contents of a file to the local processor, allows it to be modified, and replaces the copy on the file server at the end of a session. Because IFS does not support random access files, it was not considered a candidate for extensions to support database applications.

The Xerox Cedar environment uses the Alpine filesystem. Alpine's primary purpose is to store files that represent databases [26]. It also provides support for ordinary files containing documents and programs. Alpine uses lessons learned from the design and implementation of the Juniper filesystem. Alpine clients use Cedar's RPC support mechanisms to communicate with an Alpine server. The directory mechanism, providing mappings from user supplied filenames to the internal Alpine names, is not part of the Alpine filesystem. Instead, this naming system is itself an Alpine client.

2.2.2. Apollo Domain

The Apollo Domain operating system, *Aegis*, is a networked operating system that provides transparent access to remote files and devices [9,10]. Files and devices are shared across the nodes of a Domain system. Each file or device has a unique internal identifier that describes its location in a network of Apollo systems. Directory services map external names into one of these unique identifiers. Aegis' demand–paging support determines the proper location and arbitrates access to these resources.

2.2.3. Sun Remote Filesystems

Sun provides two (sometimes confused) disk–related network protocols. The first of these, the ND protocol, provides block–level access to remote disks. It does not implement remote filesystems. The more recent NFS network filesystem uses a finer granularity to provide read/write sharing of the same filesystem by several clients.

2.2.3.1. The Sun ND Protocol

The ND (Network Disk) protocol provides diskless workstations with access to disks on remote systems. The ND protocol provides a block level access to the raw device; the filesystem is not part of this protocol. It provides the client with a device driver that maps disk I/O requests into network messages. ND servers interpret these messages as requests to read and write specific sections of their local disks and transmit the results to the ND client. To the ND client, an ND disk is similar to a local disk. This ND device has a different set of procedures to transfer data; instead of manipulating registers on a controller to start data transfers, the device driver builds a network packet and sends it across an ethernet.

The ND server allocates sections of the local disk to individual ND clients. The lower per–byte costs and better performance of large disk drives can be shared by several workstations. While the ND protocols allow several workstations to share the physical disk drive, the data on the disk drive is not shared. Each client has exclusive read/write access to a portion of the physical disk drive. Clients can share portions of the drive in read–only mode.

Diskless workstations can use the ND device driver model to boot from remote disks. Some other network filesystems require local disks to boot individual processors. These disks usually provide little storage space and relatively slow transfer rates.

2.2.3.2. The Sun NFS Network Filesystem

The SUN Network File System extends the UNIX Filesystem onto a network [15,85]. NFS servers export filesystems; NFS clients mount these filesystems. A server allows many clients to access the same filesystem. Clients can perform read and write operations on the filesystem; the NFS protocols keep the filesystem in a consistent state.

The NFS design uses stateless servers. State information, like file offsets and inode information, is sent to the client after each operation. The client presents this information for later transactions. This approach has several advantages. Because the client holds the state information, servers can crash and reboot without disrupting service (although clients are delayed until the server recovers). Thus, the NFS protocols are robust across certain failure modes. Because servers maintain no state information, they can handle arbitrary numbers of clients — the server does not keep any per–client tables. Servers can handle many clients with low traffic levels as well as a small number of clients generating high traffic levels.

To guarantee that write operations have completed, NFS servers use synchronous I/O to the local disk. When writing to an NFS server, the client blocks until the transfer is complete. This has a significant negative affect on performance. Some NFS implementations allow asynchronous writes in their servers [86]. This change can more than double writing throughput for applications that transfer large amounts of data. It also means that some types of server failures leave files partially updated and provides no indication of this to the client.

The NFS protocols do not provide file locking nor do they guarantee that each write is atomic. File locking requires that the server maintain state information. Additional protocols, in parallel with NFS, do provide file locking primitives on NFS partitions [82].

Large write operations on a standard UNIX filesystem are a single, atomic operation. Large writes may require several network transactions between the NFS client and NFS server. Because the server does not maintain state, the single large write operation is broken into several consecutive operations. It is possible for several clients to interleave their requests in such a way that the final contents of the file are a combination of data from the clients.

2.2.4. IBM Remote Virtual Disks

The IBM Remote Virtual Disk protocol provides block–level access to remote disks. This protocol provides functions similar to those provided by the Sun ND protocol. It has some additional functions that manage protections on disk partitions [18].

2.3. The Distributed System Model

Distributed Systems combine networks of machines into what gives the appearance of a single system. They usually implement some form of network filesystem; files and devices generally are available from any processor in the system. The same mechanisms provide access to these resources from all processors, giving the system a measure of *location independence*. Location independence implies implicit access to remote resources; users need not use different constructs to access resources connected to a non–local processor. These systems are characterized by the following features:

- Files, devices, and (to a lesser extent) processors are accessible regardless of their location.
- Applications are bound to a processor at execution startup. In some cases, an application can be moved among similar processors.
- A distributed system can survive the failure of one or more component systems. When these components fail, portions of the system become inaccessible. Access to the remaining portions of the system continues uninterrupted.
- Distributed systems can be augmented with additional component systems. This increases the total aggregate computing bandwidth of the system and allows the system to support more simultaneous users.

The first system discussed, LOCUS, combines its many component processors to provide the image of a single centralized system [72,84]. The components of a LOCUS system share the same view of the resources in the system. A single name refers to the same LOCUS resource regardless of which processor interprets the name.

The second system discussed, the Newcastle Connection, does not bind the individual systems as tightly as LOCUS [27]. Instead, it provides mechanisms that allow the application program to access resources on remote nodes without any syntax changes. Newcastle Connection systems have separate views of the combined filesystem; when presented to different machines, the same name can refer to different objects in the filesystem.

Both of these systems provide access to remote resources. Such systems allow applications to run on powerful processors while using resources connected to workstations. However, the entire application executes on the supercomputer. These systems provide a coarse granularity of processor sharing that does not meet our criteria.

2.3.1. LOCUS

LOCUS integrates several (possibly heterogeneous) computers into a single system [72]. It extends the familiar environment of the UNIX timesharing system into a multi–computer environment. It provides an environment that simplifies the development of distributed applications.

LOCUS extends the UNIX environment onto a network of computers. Processes share a common filesystem, regardless of their assignment to CPUs. Processes communicate with the same mechanisms used in a UNIX system. Distributed applications can be modeled as a set of UNIX processes. The LOCUS operating system resolves issues of process location, network communications, and filesystem operations.

LOCUS provides the same granularity of sharing as the UNIX system — that of the UNIX process. Processes are started on a single CPU; the system can move them to other CPUs of the same architecture. LOCUS does not provide for the same process to execute on multiple processor architectures.

2.3.2. The Newcastle Connection

The Newcastle Connection is a user–level implementation of UNIX United [27,37]. Russo's thesis describes a kernel implementation of UNIX United [77]. UNIX United provides a framework for connecting the filesystems of individual UNIX systems into a larger hierarchy. The roots of each UNIX system are directories in this extended tree.

These extensions allow applications programs to reference remote files and devices without modification. To write to a remote tape drive, the user might use the pathname

/../unixb/dev/rmt0. The leading */../* indicates to the pathname resolution code that the file is in the root's *parent* directory, then down into the *unixb* directory. The *unixb* directory is actually the root directory of another UNIX system named unixb. The UNIX directory structure can be used to group the systems of various departments into appropriate groups. A user in the Electrical Engineering Department at the University of Illinois might access the password file on a machine in the CS department with the pathname */../../cs/a/etc/passwd.* To access the password file on a machine within the CS department at Purdue, he might use */../../../purdue/cs/mordred/etc/passwd.* This scheme provides an infinitely extensible naming tree above the local roots of each system. However, Newcastle Connection names are not location independent. The user must have knowledge about the meta-structure of the tree combining systems and knowledge about current node's location in this meta-structure.

In addition to allowing access files and devices on remote systems, UNIX United provides for program execution on remote processors. This can allow for faster execution of programs by specifying that they run on faster or less loaded CPUs. The mechanism for specifying the CPU to execute a program is tied to the system where that binary resides. If a binary exists on system A, it executes on system A. The UNIX pipe construct can be used to build series of connected programs and execute them in parallel on separate processors. The UNIX text processing stream makes a good example of this feature:

```
tbl | eqn | pic | ditroff | d300
```

By specifying program images on separate hosts, the separate processes can be scheduled on 5 different processors. This provides the potential for a five-fold throughput increase; the actual improvement is somewhat less due to the communications costs between processors and the synchronization that occurs at the original hosts [37,77].

Like LOCUS, the Newcastle Connection extends the scope of a UNIX process. Processes can now access remote resources. However, CPU sharing still occurs at process granularity. Individual processes can not execute on multiple CPU architectures.

2.4. The Client/Server Model

The client/server system is another form of distributed operating system. These systems use a different approach to provide services traditionally implemented by the operating system kernel. Client/server systems demonstrate a trend toward reducing the size of the operating system; the major function of these operating system kernels is to provide message passing between processes. *Server* programs provide the services provided by the kernel of more traditional operating systems. To the (new) kernel, these server programs are additional user applications which might be located on non–local nodes. Processes wishing to use services such as filesystems are clients of these servers [22,30,41,74,75].

A result of this approach is that much of the operating system's overhead can be moved to another processor. In some cases, the operating system overhead can consume 50 percent or more of the CPU cycles. When most of the operating system overhead is removed, application programs can achieve higher sequential processing speeds on that CPU. Communications between the application and the traditional operating system services are now more expensive: the new arrangement introduces communications costs between the application and the operating system.

Client/server systems are often implemented in environments with many processors. To exploit the parallelism available across these processors, an application program must be partitioned manually into concurrent processes. Client/server systems concentrate on providing efficient communications mechanisms between processes and processors because the system performance is so dependent on this underlying communications mechanism. Slow communications mechanisms can reduce the system throughput dramatically.

The subsections below describe several systems based on the client/server model. Each provides the programmer with the ability to partition application's into separate pieces to execute in parallel on separate processors. All of these systems can execute several instruction streams from the same logical address space. In many cases, the degree of concurrency is limited to the number of processors sharing physical memory. This allows reduced solution time for some appropriate algorithms in systems configured with multiple processors sharing a single memory.

2.4.1. Amoeba

The Amoeba operating system is a client/server system being developed at the Vrije University in Amsterdam [81]. The Amoeba system comprises four component types:

- Workstations, one for each user,
- Pools of processors, dynamically assigned to user tasks,
- Specialized servers (e.g., file servers, bank servers), and
- Gateways which link Amoeba systems at separate sites.

The Amoeba kernel provides message passing facilities and few other functions. This is motivated by the desire to keep the kernel small, enhance its reliability, and provide the traditional kernel features in user processes to facilitate flexibility for experimentation.

Amoeba assigns tasks to processors at process creation time. It is possible for several processes to share the same text and data segments though the individual processes each have private stack segments. Processes which share text and data share the same processor. Because applications can dynamically acquire and release processors, the aggregate bandwidth available to an application can be quite large. Applications must still be explicitly partitioned properly to take advantage of this bandwidth.

2.4.2. Stanford V System

The V kernel project at Stanford University provides services in an environment of diskless workstations connected by a high-speed local network [30]. The V kernel provides a fast, message-based communications framework for RPC among clients and servers. *Teams* of one or more processes sharing a single address space provide processing power for a single job.

V System message passing is synchronous. After sending a message, the sending process blocks until the recipient replies. Since the V kernel guarantees delivery of messages, programmers need not implement protocols to ensure reliable communications between processes. Within a team, starting a new process is relatively inexpensive. These inexpensive processes provide a means to implement asynchronous communications; many programmers create inexpensive processes only to send a message, wait for the acknowledgement, and then terminate.

Each V process resides on a single logical host; all processes in a V team live on the same logical host. A logical host resides on exactly one physical host though a physical host may support many logical hosts.

Logical hosts can be moved between physical hosts, allowing a set of processes to be moved from a busy processor to an idle processor; the busy processor suspends the processes on the logical host, creates a logical host on the target processor, and sends the state of the logical host to the remote host. The actual implementation allows the logical host to execute while its state is being shipped to the new host. After the bulk of the state is transferred, the V kernel checks to see what state has changed, and sends updated information for those parts of the logical host [83].

Adding more processors to a V system can improve aggregate throughput. Decreasing the solution time for a single application requires explicit restructuring of the application or using faster processors. Cheriton has developed a master–slave approach to structuring applications to execute on networks of personal workstations that do not share memory and have limited communications bandwidth [29].

2.4.3. CMU MACH

The CMU MACH system provides a new foundation for future UNIX development [22,74]. MACH draws heavily on previous experience with Accent and uses a similar underlying model [75]. MACH is designed to provide the facilities needed to exploit general–purpose, shared–memory multiprocessors.

The MACH address space (*task*) can have many active instruction streams (*threads*). There can be many such tasks in a running MACH system. MACH's multiple–thread model provides a foundation for the easy use of parallel algorithms; shared–memory provides inexpensive data transfer between the threads of a task. On multiprocessors, MACH schedules the threads concurrently and the application runs faster. On uni–processors, MACH schedules the threads sequentially. Although execution on a uni-processor takes longer, the program produces the same results.

MACH provides programmers with constructs to partition a program into parallel threads of computation that share a common data space. MACH also implements parallelism without a shared data space; such a case is a simple RPC implementation using the MACH message passing facilities between tasks.

MACH provides a good base for a CLASP implementation. The new virtual memory management system allows non-kernel tasks to be specified as the paging mechanisms for individual tasks [22]. This allows processes to page to user filesystems; MACH does not require disk partitions to be dedicated to swapping [22]. The user level routines for CLASP should be easily ported to the MACH environment. The page-management routines for CLASP can be implemented using a non-kernel task that is assigned paging responsibilities for a process.

2.5. The Multiprocessor System Model

Multiprocessor systems are another way to share resources. In these systems, a number of processors are connected to a common memory. These processors can share peripherals. An multiprocessor system with N CPUs can usually provide almost N times the performance of a single processor system. Because the CPUs share the same set of peripherals, an N processor system can cost much less than N times the cost of a single processor system.

Multiprocessor systems provide increased aggregate throughput. Because they share memory, these systems can balance the load on each CPU without incurring high costs to move jobs across a network between systems. With the appropriate compiler and operating system support, these systems can reduce the execution time for applications.

The first set of systems described below uses many processors to improve aggregate throughput. In these systems, the operating system allocates jobs to idle processors from a single ready queue. Jobs may execute on different processors from one timeslice to the next. Because moving jobs between processors does not incur any communications costs, a single multiprocessor system with N processors provides better aggregate throughput than a network of N uniprocessor systems (which have overhead when moving tasks between systems). Idle processors can execute any ready job because all jobs are in a shared memory. By themselves, these systems do not reduce the solution time for a single application. However, applications can be restructured to exploit the concurrency available in these systems and reduce their execution time.

The second set of multiprocessor systems uses compiler technology to detect implicit concurrency in sequential code and constructs suitable for vector operations. Compilers for these systems break sequential code into blocks that can be scheduled to execute concurrently by the operating system. Data dependencies between blocks of code determine when blocks can be scheduled. These dependencies are similar to Petri nets; when the input values are ready, the block *fires*[71]. These systems often use hardware technology similar to the first set of multiprocessors. This hardware reduces memory contention and saturation of shared resources through complex and expensive memory hierarchies. Some of these systems reduce contention for the global memory by providing each processor with a private memory.

The multiprocessor systems described in this section comprise single logically and physically integrated systems; failure of critical elements in these systems stops all operations. They are like centralized timesharing systems in this respect, although they might offer higher performance or easier expansion routes than a centralized, single–processor timesharing system. These systems also do not support the enhanced interfaces (e.g., mice, bit–mapped displays) available on workstations. It currently is not cost effective to support these interfaces on these machines for many users. Multiprocessor systems are more expensive than single user workstations. While some multiprocessor machines can be expanded inexpensively to include more processors, there is still a significant initial cost.[5]

In addition to the systems we are about to describe, other multiprocessor systems include the Cray–X/MP, the CDC Cyber series, and the DECsystem 20. The DEC VAXcluster is a closely–coupled system that allows multiple processors, with separate memory, to share access to common peripheral devices. The VMS operating system uses this hardware to provide load balancing and improved system reliability [57].

The next sections describe several multiprocessor systems in more detail. The first section describes the C.mmp project, an early experiment in multiprocessor systems. The second section describes the Sequent and Encore systems, where the processors share a single global memory. The third section describes the University of Illinois CEDAR project, which combines multiprocessor systems with compilers that automatically restructure sequential applications into concurrent applications.

[5] A 2 CPU, 2 Mbyte Sequent Balance–8000 system costs $59,000. A 4 CPU, 4 Mbyte Sequent Balance–21000 system costs $139,900. The Balance–21000 has backplane slots for more processors than the Balance–8000. Additional processors are $16,000 per card (2 processors). An Alliant FX/8 with 1 CE costs $150,000. A fully expanded FX/8 (with 8 CEs) costs $450,000. A SUN–3/52 workstation (with a local disk and cartridge tape drive) costs $13,900. A diskless SUN–3/50 costs $4,995.

The Sequent prices are as of August 1986. The Alliant prices are from June 1985. The SUN prices are from fall 1986.

2.5.1. C.mmp

The C.mmp amd Cm* systems provided an early testbed for experiments involving multiple processors connected to a common memory hierarchy.

The C.mmp system connects 16 PDP–11 processors to a large common memory through a crossbar switch [88–91]. Address relocation boards in each processor map virtual addresses on the private bus to physical addresses in the shared memory. Thus, each processor in the C.mmp system could access the entire common system memory. Private, per–CPU memory is used for off–line maintainence and certain private operations.

The Cm* machine contains clusters of processors. Like the C.mmp system, each processor has a private memory. However, Cm* has no single global shared memory. Instead, each cluster of 5 machines is connected with a special *Kmap* processor that allows the CPUs to access the memory connected to other CPUs. Kmap processors communicate between themselves to implement cross–cluster memory accesses. Each Cm* processor can access any of the physical memory using a uniform method, but the access costs vary depending on the relative locations of the processor and memory. Access times for memory in other clusters can be as much as 10 times slower than access of the memory directly connected to the processor [68,69].

2.5.2. Encore and Sequent

The Encore and Sequent systems represent a family of systems that use *timeslice sharing* to allow a fine partitioning of the available computing cycles provided by many processors to many processes. In *timeslice sharing* systems, a job may execute on a different processor each timeslice. These systems use many processors attached to a common memory to improve aggregate throughput. A processor can be reassigned to a different task each timeslice. This technique allows a pair of 4 MIPS processors to provide service for two real-time processes requiring 3 MIPS and a third process requiring 2 MIPS. A pair of 4 MIPS single-processor systems can not meet the needs of the same 3 processes. Timeslice-sharing systems do not reduce the solution time for a single application. The two processor system (4 MIPS each processor) does not meet the needs of an 8 MIPS program nor does it meet the needs of a 5 MIPS program.

Explicit restructuring of applications for concurrency allows timeslice sharing systems to reduce the solution time for a given problem. The shared-memory between processors improves the performance over that achieved when the same application is spread across a network of processors. The communications costs between separate processes on these systems is much lower than the cost between processes on systems separated by a network.

The common memory provided in timeslice sharing systems can be used for an efficient data sharing scheme. Some of these systems permit processes to share portions of the address space [12,14]. Other systems, such as MACH, let processes share the entire address space. In each of these cases, the application must still be restructured to reduce the solution time for that application.

One of the advantages provided by these machines is their ability to be expanded inexpensively. If more computational power is needed, additional processors can be purchased as single boards. The additional boards plug into the system and give an immediate performance improvement.

2.5.3. CEDAR

The CEDAR project at the University of Illinois is building large scale multiprocessor supercomputers [42]. To attain this goal, the project is concentrating on connecting many processors to a common memory hierarchy, using compilers that automatically restructure sequential code into segments that can be executed in parallel, and developing a control hierarchy to coordinate these many parallel tasks.

The CEDAR compilers isolate small blocks of code that can be scheduled independently of each other [58]. The compiler decomposes loops without recurrence relations (and those with specific types of recurrence relations) into several smaller loops that can be scheduled to run on several processors concurrently.

The underlying hardware model for the CEDAR architecture is many processors sharing a common hierarchical memory and each processor having a private memory. Code and data required to execute a block are copied to the private memory. The output values are returned to the common memory. Initial implementations of the CEDAR system are planned for 8 and 16 processors. Larger systems, with 1024 or more processors, are being designed.

The CEDAR system reduces the solution time of a single application by partitioning the application into blocks that are executed concurrently. The compiling and scheduling techniques used in the CEDAR system are applicable to a range of memory hierarchies. Systems with a common bus and a single global memory can show reduced execution time by using compilers that automatically restructure the application. The CEDAR compilers show that automatic restructuring techniques can be used to improve execution time for sequential applications in a multiprocessor environment.

2.6. Summary of Related Work

We have examined a number of existing tools that provide support for resource sharing. However, they do not meet the criteria defined in chapter 1.

The Remote Procedure Call allows applications to be partitioned between client and server systems. RPC often requires changes to source code. All communications between the RPC client and server must be through the defined procedural interfaces. Applications that pass information using global variables or that use pointer-based structures can not be partitioned transparently. RPC implements a restricted subset of the procedure call abstraction. The NDS architecture removes some of the RPC restrictions by providing a shared data segment and allowing client and server routines to access shared global variables. NDS does not support sharing of variables stored outside of the data segment (e.g., variables stored in the stack). Both RPC and NDS store the client and server portions of a program in separate files.

Object-oriented systems support remote operations. However, these systems require that the application be coded using the appropriate languages. A large FORTRAN code would have to be recoded in the appropriate new language to take advantage of these features. This violates our rule that we will not require restructuring or recoding of applications.

The network filesystems presented in this chapter allow applications to run on any machine. They are no longer restricted to executing on the system that stores the application's data. Because network filesystems only provide access to data on remote processors, they do not solve the problem of partitioning an application so that it executes faster. Applications can be moved, in their entirety, to faster processors. But, this does not satisfy our criteria that the workstation must be used for more than submitting jobs to a fast processor.

Like network filesystems, distributed systems provide access to local and remote resources. Applications can be moved to larger, faster processors in these systems and execute faster. These systems move entire programs to other processors. Like network filesystems, they fail the criteria that the local workstation be used for more than submitting jobs to a faster processor.

The final set of systems described, the multiprocessors, can provide improved performance. They provide hardware support for compilers that analyze sequential programs to generate concurrent programs. Because of their relatively high costs, these systems are more appropriate for compute servers than as workstations or clients.

CHAPTER 3.

CLASP AND THE CROSS ARCHITECTURE PROCEDURE CALL

The CLASP architecture is designed to provide *compute servers* for workstation users. The compute servers are connected to the workstations using local area networks like Ethernet. To meet our definition of effective, these compute servers must:

- exploit the speed difference between client and server processors,
- require no changes to application code,
- continue to use the workstation for appropriate portions of the computation,
- have no impact on the use of workstation–specific interfaces and devices such as bit–mapped displays, mice, and windows.

The CLASP architecture implements the traditional UNIX process model on a new underlying foundation. This new foundation allows appropriate sections of an application to execute on the most appropriate processor. Like the Remote Procedure Call and NDS architectures, the CLASP architecture partitions applications at the procedure call level. CLASP differs from these architectures by providing a more transparent mechanism to access remote procedures. CLASP also removes some restrictions imposed by these other architectures. Procedure calls that cross architectural boundaries are named *Cross Architecture Procedure Calls* (or *CAPCs*) in the CLASP software architecture.

CLASP allows a set of routines within an application to be accelerated by recompiling them for a faster processor. They will be executed on that processor. CLASP manages the movement of data and the control thread between processors. CLASP seldom requires source code changes to exploit the speed advantages of the compute server.

A new CLASP loader collects object files for multiple processor architectures into a single executable file. The loader groups instructions for a particular processor architecture into a contiguous virtual address range. The resulting executable file contains information that describes the processor architectures required to execute different portions of the address space.

A set of modifications to the operating system kernel handle the runtime details of the CAPC software architecture. These modifications handle the detection of accesses to code for non–local architectures, the transfer of control to a processor of the correct architecture, and the transfer of pages in the address spaces between processors. These operations are transparent to the application program; the kernel implements all of these operations.

An application's performance is controlled by the way it is partitioned between the client and server. Some partitionings yield better performance than others. It is not efficient to use CAPCs to calculate a square root; the network overhead overwhelms any speed advantages provided by the remote processor. An application is partitioned by compiling specific source files for specific architectures.

This chapter describes the characteristics that the workstation and compute server must share. It describes the virtual address space of a CLASP process. One section describes how CAPC control transfers are detected and processed. Another section describes process state manipulations, including changes in the virtual address space and paging between systems. The chapter closes with a comparison of CAPCs and RPCs.

3.1. Required Homogeneity

The CLASP architecture requires that processors share data representations and have a common address space. However, each processor may have different instruction sets, register sets, and stack frame formats.

3.1.1. Homogeneous Data Representation

CLASP requires that all participating processors agree on how data is represented. Fortunately, many processors share a common data representation. For example, the Motorola 68000 and IBM RT–PC microprocessors have the same data representation as the Convex C–1 and Alliant FX computers [1,4,7,11,47,48]. Because they share the same internal data representation, these processors can exchange information without intermediate format conversions. The processors can move data as a series of bytes without interpreting the contents and without providing type information.

When compiled and executed on different processors with the same data represenation, a subroutine generates the same output. We are not concerned with the actual machine instructions that implement the subroutine, but rather with the relative processing rates of the two processors. One processor may execute the subroutine signficantly faster than the other. For example, a CPU with vector instructions might execute a matrix multiplication subroutine significantly faster than a CPU without vector instructions. Each processor executes a different sequence of instructions at different rates, but the generated output is the same in both cases.

3.1.2. Homogeneous Address Space

Many languages provide pointer data types in addition to character, integer, and floating point data types. A CLASP system must provide a homogeneous data representation for pointer data types. To provide support for pointer types in a CLASP system, the processors must share a common virtual address space. The common virtual address space allows the client and server processors to share pointer data types. Therefore, CLASP implementations for languages that provide pointer data types require that client and server processors have an intersecting virtual address space. An exact address space match is not necessary; a CLASP system can function with a sufficiently large intersection.

Some application languages, like FORTRAN, do not provide pointer data types. For these languages, a common virtual address space is not necessary. Instead, a direct mapping between the two address spaces is sufficient. Modifications to the loader to handle the skew in the address space allows these languages to be split between client and server processors. The loader adjusts operand offsets to accommodate the different locations in the virtual address space.

3.1.3. Homogeneous Compilers

Identical data representations and intersecting address spaces provide a base for the CLASP software architecture. To use this system, the compilers for both architectures must share several properties. In particular, variable types (e.g., int, long, float, etc.) must translate to the same length, alignment of variables within structure or record definitions must be the same, the compilers must use similar argument passing techniques, return values must also be compatible. Many UNIX implementations include C compilers based on the portable C compiler and share many of these properties [52].

High–level programming languages provide several basic data types (e.g., *int*, *float*, *short*, *long*, etc.) and mechanisms for constructing more complex data structures (e.g., the PASCAL *record* or the C *struct*). Both compilers must map simple data types to the same sizes. A compiler that translates *int* into 2 bytes of storage is incompatible with a compiler that maps *int* into 4 bytes of storage. For more complex data types (e.g., *record* or *struct*), the compilers must obey the same alignment restrictions. Some compilers align to even byte boundaries to meet processor restrictions. Other compilers align to 4 byte boundaries to match memory subsystems. A CLASP system requires that the compilers use the same alignment rules.

The CLASP routines must be able to pass a subroutine's argument vector to the remote processor. In many cases, argument vectors are stored as a contiguous array of bytes. The first parameter is stored at the low address of the byte array and the last parameter is stored at the high end of the array. Other compilers place the first few arguments in registers and store the remaining arguments as an array of bytes [67]. For these cases, CLASP kernel routines collects the register arguments and the additional arguments into a single vector to pass to the remote processor.

The CLASP routines must also be able to collect the return values. Many compilers place return values in one or two registers. On the Motorola 68000, return values of 32 bits or fewer are passed in the register D0. 64 bit return values are passed in D0 and D1.

Routines that return complex data types are handled in several ways.[1] Some compilers

[1] This difference prevents us from using the IBM RT with its current compilers as a client of processors such as the SUN, Convex, or Alliant.

return a pointer to an instance of the complex data type [52]. This instance is in a static per–routine buffer. The calling routine copies the instance to its desired location. Other compilers use a different approach [49,67]. The calling routine prepends an extra argument to the parameter list. At call time, this parameter contains a pointer to an instance of the complex data type. The called routine stores its results in the supplied buffer.

CLASP can use compilers that implement either technique, but both compilers must implement the same mechanism. Compilers that change the argument list according to the return value of the routine can not be easily matched with compilers that do not.

3.2. The CLASP Address Space

CLASP provides each application with a single address space. The client and server portions of an application share this address space. Individual pages of the address space reside in the physical memory of one processor or another. When a processor requires a page that resides in the remote processor's physical memory, the data is demand–paged across the network and placed in the local physical memory. The page tables on both processors are modified to reflect the page's new location.

The address space contains the instructions for both processor architectures. For each architecture, the CLASP loader consolidates the instructions into a single range of addresses in the address space. A CLASP executable file contains a table describing the instruction space for each architecture. The loader generates this information when it builds the executable image. The CLASP kernel uses this instruction space information and the virtual memory protection hardware to detect CAPC calls and returns.

The CLASP address space is similar to the UNIX address space. However, the single text segment of a UNIX process is replaced with a number of text subsegments. Each text subsegment corresponds to a processor architecture. The instruction space table in the executable file describes the boundaries and architectures of these text subsegments. Figure 3.1 shows the layout of the multiple text segments, the data segment, and the stack segment for the normal UNIX process and a CLASP process.

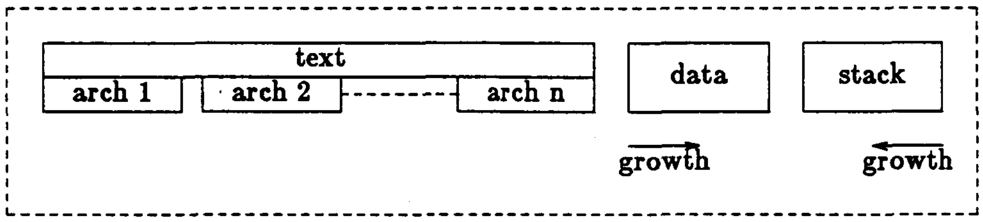

Figure 3.1
CLASP Address Space Layout

The UNIX stack segment contains the activation records for called procedures. Each of these activation records is in the format of the local processor. The CLASP stack segment contains activation records for both processor architectures. CLASP inserts its own information between adjacent activations records for different processor architectures. This information allows CLASP to handle control transfers between architectures. This is described in more detail in section 3.2.2, *The Clasp Stack Segment*, and section 3.3.2, *Stack Frame Manipulation*.

3.2.1. The CLASP Text Segment

The text segment contains the executable code for the client and server architectures. The loader combines the instructions for a particular architecture into a *text subsegment*. The CLASP loader aligns the text subsegments on virtual memory page boundaries. This lets the CLASP kernel use the virtual memory protection mechanism to prevent a processor from fetching instructions from another architecture's text subsegment. Some virtual memory systems provide an execute protection bit. Other virtual memory systems combine read and execute permission into a single protection bit. In the former, a processor may be allowed to read the instruction space for remote architectures. Systems with a single protection for read and execution permissions make it impractical to provide read permission for the pages that are non-executable.

Each processor maintains its own page tables for residency and protection information about the address space. The protection information is used to detect and process subroutine calls that cross CPU boundaries. The client sets its page table entries for the server architecture instruction space to disallow execution.[2] Attempts to execute instructions from those pages on the client CPU generate traps to the operating system kernel. The server allows execution of the instructions appropriate to its architecture, but protects the pages with client architecture instructions to prevent execution of client instructions on the server processor.

[2] Some virtual memory systems provide read, write and execute protection bits for each page. For these systems, CLASP clears the execute permission bit. Other VM systems combine read and execute permissions into a single bit. To disallow execution in these systems, CLASP must also disallow read permission on those pages.

3.2.2. The CLASP Stack Segment

Like the UNIX stack segment, the CLASP segment contains subroutine activation records. The CLASP kernel transparently extends the stack segment to hold new subroutine activation records or stack frames. The CLASP system stores activation records for both processor architectures on the same stack. When a CAPC causes the control thread to move between processors, the CLASP kernel arranges for the remote processor to have the correct stack frame for its activation record. The kernel builds a CAPC packet to describe the transformation between the processor architectures and stores this information on the stack.

This CAPC frame contains the argument vector location and length, the called procedure's entry point, the return address for the calling procedure, and the stack bounds. CAPC calls and returns use this frame to exchange information. The remote processor may replicate some portions of the procedure call information if needed. For example, the VAX hardware architecture provides an argument pointer register that can be loaded with the location of the original arguments. The Motorola 68000 family does not provide an argument register, instead it expects the arguments to be just above the stack pointer at procedure entry. A 68000–based CLASP system copies the argument vector to set up the correct stack layout for a procedure call. Some hardware architectures require the CLASP kernel to replicate the procedure call information. The original copy is stored in the client's format while the second copy follows the server's stack protocol.

The local processor's general registers are not kept in the CAPC frame. There are several reasons not to store the registers. The remote processor uses its own registers, it will not overwrite any registers on the local processor. The register layout may differ between processors; the remote processor might have more registers or they may have different names and semantics. The 68000 uses register A7 as a stack pointer; the IBM RT uses register R1 as its stack pointer.

Figure 3.2 depicts the stack segment as it looks while executing a routine on the remote processor. Procedures 1 and 2 execute on the client; procedures 3 and 4 execute on the server.

A processor can examine variables stored in the stack segment by the peer processor. The variables have the same internal representation; the remote processor requires only the address of the variable to access, and modify, the variable.

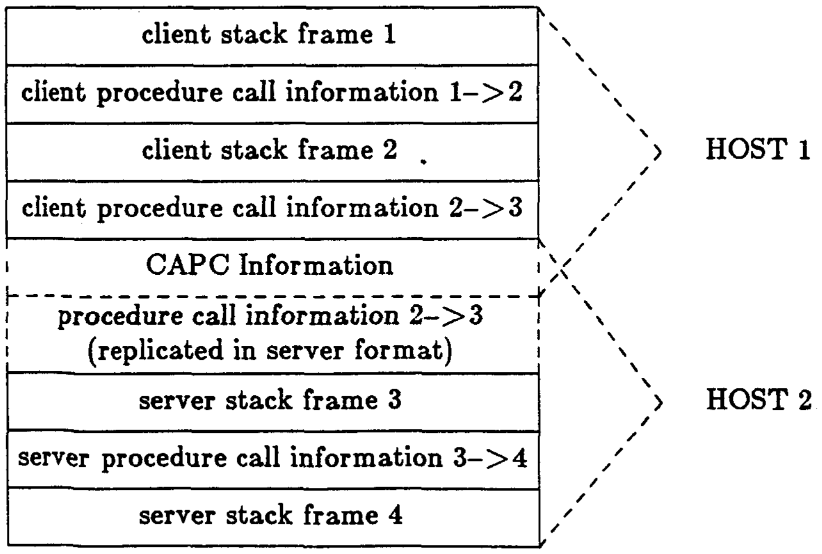
client stack frame 1
client procedure call information 1->2
client stack frame 2
client procedure call information 2->3
CAPC Information
procedure call information 2->3
(replicated in server format)
server stack frame 3
server procedure call information 3->4
server stack frame 4
HOST 1
HOST 2

Figure 3.2
CLASP Stack Layout During a CAPC

In a single architecture model, programs are able to destroy their activation records. Because the CLASP kernel stores the CAPC frame in the user's stack segment, these faulty programs also can destroy the CAPC frame.

3.3. CAPC Linkage

This section of the thesis describes how the CLASP kernel processes CAPC calls and returns. To process a CAPC call, the kernel must detect the reference to a non–local procedure, identify the argument vector's location and length, determine the return address, and determine the stack boundaries. After gathering this information, the kernel builds a CAPC frame on the user stack and transfer control to the server. The CLASP kernel on the server uses the CAPC information to build a calling frame in the server architecture's format and to start execution of the remote procedure.

This section also describes the mechanisms used to implement nested CAPC calls. The CLASP system allows server routines to invoke routines on the client.

3.3.1. Detecting Calls to Procedures for Other Architectures

The CLASP kernel uses the virtual memory protection system to detect CAPC calls and returns. The CLASP loader generates a map describing the architecture for each of the text subsegments. The kernel uses this to protect the instructions for non–local architectures. When an application makes a procedure call to an address in one of these protected regions, the virtual memory system generates a fault and traps to the kernel. The kernel examines the faulting instruction to determine whether it is a call or return. The target address for the call or return is the address that generated the fault.

For a CAPC call, the kernel also determines the location and length of the argument vector. This is an architecture–dependent operation. The kernel places this information in a CAPC frame and stores it on the stack. CAPC frames contain:

- a location to start execution,
- the location of the arguments to the subroutine,
- the count of arguments to the subroutine,
- the stack pointer, indicating where the server can build a procedure call frame,
- information from the server used to restore the stack frame upon completion of the CAPC, and
- the arguments, if their total size is less than 64 bytes.

The kernel sends the address of the CAPC frame and a copy of the CAPC frame to the remote processor. The kernel on the remote processor uses the CAPC frame to build an activation record for the called procedure. After building the activation record, the kernel starts the called procedure at its entry point.

When the remote procedure finishes, it returns to the address stored in its activation record. This address is in the text segment for the client processor. The virtual memory system generates a protection fault when the processor attempts to fetch the instruction at the return address. The kernel determines that a return instruction caused the fault. The server loads the return values into fields in the CAPC frame. After loading the return values, the server passes the CAPC frame to the client processor. The client processor retrieves the return values from the CAPC frame, loads them into the appropriate registers for the client architecture, removes the CAPC frame from the stack, and resumes execution of the application code.

The stack frame after several nested CAPCs might look like the frame depicted in figure 3.3. This stack frame is for a process with procedures P1 through P5. Each procedure calls the next higher numbered procedure. The client processor executes procedures P1, P2, and P5. Procedures P3 and P4 execute on the server architecture.

Many programming languages provide a mechanism to pass functions as formal parameters to other routines. The UNIX *qsort*(3) routine uses this mechanism to allow users to

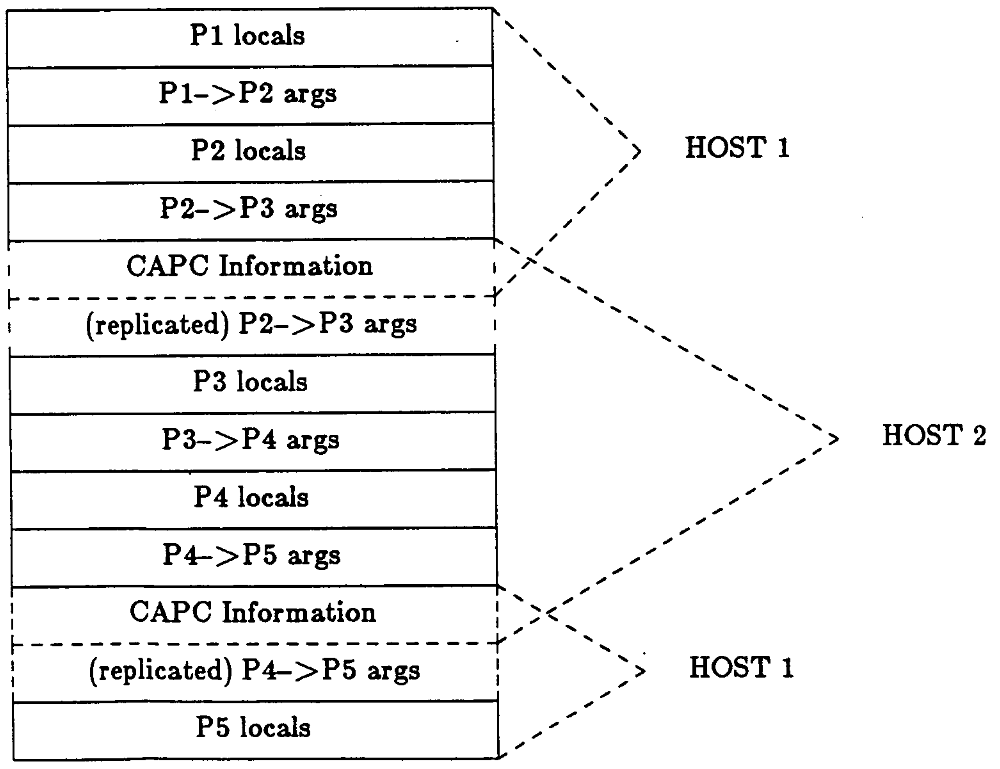

Figure 3.3
Stack Layout During Nested Cross-Architecture Calls

specify a function that returns the ordering of two elements. CLASP allows applications to pass both client and server procedures as formal parameters. The procedure's architecture does not change how the procedure is passed. When the routine is invoked from the called routine, there are no special instructions that differentiate between local and remote procedures. Appendix C contains an example program that demonstrates this feature of the CLASP architecture.

3.4. Compute Servers

When starting a CLASP process, the operating system starts any server processes that might be required by the client. The CLASP kernel starts these processes during the UNIX *exec*(2) system call. If the kernel can not instantiate a server, the *exec*(2) call is aborted. These failures are mapped onto an existing failure condition for the *exec*() call. Failed server instantiations are reported to the caller as ENOMEM errors, a message that indicates the system was unable to allocate swap space for the new executable file.

For some processes, a program may never invoke the routines implemented on the server. In these cases, the server startup cost has been wasted. However, delayed server instantiation introduces new failure modes into an application program. If the kernel delays the instantiation of the server until the application attempts to execute code for that architecture, applications must be prepared for a potential error on any procedure call. By establishing servers at process creation time, we do not introduce any extra failure modes. The execution time saved by delaying server instantiation does not justify the programming costs to accommodate the new failure modes.

3.4.1. Selecting A Server

The CLASP kernel selects the processor that acts as a server for CAPC applications. Different instances of an application can use different server processors. Many clients can select the same server processor.

Our CLASP implementation stores a list of <*architecture,address*> pairs to describe available servers.[4] After determining the required server architecture for a CLASP process, the kernel searches this table for an entry with the correct architecture.[5] At this time, the kernel attempts to connect to the CLASP server daemon listening at the specified address. If the local kernel does not receive an answer, the *exec*(2) call is aborted. If the local kernel establishes a connection with the server process, it sends a message describing the address space: the text subsegments, their architectures, and their address ranges. The local kernel then allows the application program to begin execution.

3.4.2. Distributed Virtual Address Space

RPC systems package and ship entire parameter lists to the server on each call. For large argument lists (e.g., an array of simultaneous linear equations), this operation incurs a significant cost. Later operations (e.g., solving a linear system for a particular right hand side) require the factored array to be re–transmitted. Thus, an RPC system to solve the linear system $\mathbf{Ax} = \mathbf{b}$ might:

[4] Note that the local processor can be the server. This aids testing, but does not improve performance.

[5] The architecture information — which architectures are required and their address ranges in the text segment — are stored with the header information in the executable file.

- Transmit the **A** matrix to the server, where it is factored,
- Return the factored **A** matrix to the client, and
- Transmit the factored **A** matrix and a **b** matrix to the server, to obtain a solution.

The CLASP architecture uses demand paging to reduce paging traffic between the client and server systems. The CLASP kernel sends pages to the remote processor only when the remote processor attempts to access that page. Pages accessed exclusively by one processor stay on that processor. Thus, global variables accessed by a server routine will be demand paged to the server when it is accessed, instead of being pre-packaged and transmitted as part of the procedure call.[6]

After a call to the compute server, a set of pages resides on the server. These pages remain on the server; the server does not send the pages back to the client until the client requests them. This approach follows Denning's guidelines for working sets:[31]

> The fundamental strategy advocated here — a compromise against a lot of expensive memory — is to minimize *page traffic*.

Our implementation provides for a single copy of each page in the virtual address space. For read/write pages, this approach works well. For read-only pages, this approach is inefficient when both processors access those pages.

More recent work at Yale has implemented mechanisms for maintaining memory coherency in a loosely-coupled multiprocessor system [61]. Li's thesis provides mechanisms for replicating pages accessed in read-only mode. When pages are written, the system invali-

[6] By dereferencing pointers and including the underlying objct, RPC systems sometimes generate less network traffic than CAPC systems. For further information on this, see section 3.6 *CAPCs vs RPCs*.

dates the extra copies of that page. He outlines several protocols to ensure that only one processor has write permission on any page and describes how the rights for a page can be passed between processors. Li's research used loosely coupled homogeneous processors to run concurrent algorithms using a network of workstations. However, the results can be used to reduce network paging traffic in the CLASP system.

3.5. Process State Manipulation

This section of the thesis describes how CLASP affects a process's state. A process's state includes its address space, the extent of its address space, its file descriptors, and other information stored in the kernel. Address space information includes the boundaries of the text, data, and stack segments. It also includes the contents of those segments.

3.5.1. Address Space Bounds

The CLASP client and server processes share the same virtual address space. To ensure a consistent view of this address space, the CLASP kernel must arbitrate access to pages and the bounds of the address space. Section 3.4.2 explains the CLASP page management scheme. This section describes how the CLASP kernel manages changes in the address space bounds.

The stack segment grows dynamically to hold extra procedure call/return information. The *break*(2) UNIX system call extends, or reduces, the size of the data segment. The client and server processors must inform each other about changes in the boundaries of the address space. Each CLASP process must be able to access the same portions of the virtual address space, regardless of which processor executes the instructions.

Changes in the address space bounds are passed to server processors when the control thread is transferred to that machine. The server does not need this information until the control thread moves to the server.[7] The client processor includes the current address space bounds with the CAPC call packet. When the control thread moves to the server, the server adjusts its page maps and address space boundaries to match the boundaries presented in the client's call message.

When the virtual address space expands, the server adds new page table entries for the new pages. The server marks these pages resident on the client processor. If the server attempts to access these new pages, they will be demand paged from the client processor.

Decreasing the address space requires additional work to ensure that the client and server views of the address space remain consistent. If the client shrinks and re–expands the data segment, it replaces those pages with zero fill–on–demand pages. This operation discards any data on the affected pages. If the server has copies of those pages, it must also discard those pages. When shrinking the address space, the client must notify the server of the invalidated pages of the address space.

The CLASP kernel saves a *low water* marker for the bounds of the address space. Address space reductions set this low water marker. The client kernel passes the current bounds and the low water bounds when it passes control to the server. The server kernel examines the low water marker to see if the client discarded any pages that the server knows about. If so, the server invalidates those pages. After resolving address space reductions, the

[7] This is a design decision based on our underlying process model. Our base system (SUN Unix) provides a single control thread in each address space. In a system with multiple control threads, a different scheme to modify the address space is required. A possible scheme might designate one or more of the processors as *owner* of pages not yet in the address space. Extensions of the address space would be processed through this processor in a manner similar to the current network page fault mechanism.

server uses the current address space information in the CAPC to expand its page table to the current size. The new pages are marked resident on the client.

3.5.2. System Calls

The UNIX system call uses state information stored in the kernel. Some system calls reference simple information, such as the current time. Others reference more complex state information, such as I/O descriptors. The CLASP architecture does not attempt to replicate this state information between the client and server kernels. The client and server share only information about the bounds of the user's address space.

Applications must make system calls on the processor holding the appropriate state information. Because the client processor holds all state information, system calls occur on the client processor. User programs access UNIX system calls through a collection of C subroutines that package their arguments and trap to the UNIX kernel. An easy way to force system calls to a particular architecture is to compile these subroutines for that architecture. The current CLASP implementation uses this scheme to force system calls back to the client processor. Client calls to these subroutines execute on the client processor. Server calls to these subroutines generate CAPC calls from the server to the client processor, where the system call is executed.

CLASP does not disallow system calls on the server processor. In many cases, it can be more efficient to perform system calls on the server. For example, assume an application that uses a file stored on the servers disks. In the current implementation, I/O operations on this file are processed on the client. If a routine on the server reads from the file, the data crosses the network twice: once using the network filesystem operations from the server to the client and once from the client to the server as a CLASP network page fault. A modified version of the I/O related library routines can eliminate this overhead. These library routines can direct the I/O operation to the appropriate kernel. Other systems have successfully used this technique to redirect system calls to the appropriate processor [27,37,77].

3.6. CAPC vs RPC

The CLASP architecture improves on the RPC architecture in several respects. The best way to summarise these improvements is to note that CAPCs provide the same semantics as normal procedure calls, while RPCs provide a subset of the normal procedure interface.

RPC client and server processes execute in separate address spaces. All data required for a remote procedure call must be passed as arguments to that routine and can not contain pointers. On the other hand, CLASP client and server processes execute in the same virtual address space. This allows routines to exchange data through global variables and to share pointer–based structures such as lists and trees. Thus, CLASP procedures can be used exactly like normal procedures. Converting a program to RPC usually involves rewriting it.

There are situations where this shared address space can impact performance. When a CLASP server routine dereferences a pointer passed from the client, it may generate a network page fault to retrieve the appropriate page of the virtual address space. RPC systems dereference each pointer before packaging and transmitting the argument list. Therefore, the single RPC message contains the object described by the pointer and does not generate the extra network transaction possible in the CLASP system. Imagine the pathological case where all of a routine's arguments are pointers, scattered through the address space. An

RPC system collects all the data and sends a single large message. A CLASP system might generate a network page fault for each argument.

On the other hand, many common algorithms will be much faster in a CLASP-based system than using RPC. CLASP's demand-paging approach moves only the arguments and data that are actually used between systems. As an example, binary searches through large sorted arrays can be efficient because the accessed portions of the array are transferred to the remote processor on demand instead of prepaging the entire array to the server. Pages, once transferred to the server, remain on the server until they are required by the client processor. Pages used only by the client remain on the client; pages used only by the server will be transferred to and remain on the server. Pages of data used by both processors migrate between hosts on demand. Demand-paging allows CLASP to support arbitrarily long argument vectors. RPC systems, because they package all of the arguments into a single message, limit arguments to the maximum length of a network message.

Another improvement that CLASP makes over RPC systems is the way remote procedures are invoked. RPC systems use special calling sequences to access remote procedures. Some RPC implementations replicate this calling sequence at every place that invokes the remote routine [16]. Other implementations encapsulate these calling sequences into a local stub routine [3]. CLASP uses neither special calling sequences nor stub routines. Instead, the client uses the subroutine call instruction of the local architecture with the target address of the desired routine, regardless of whether it is local or remote. Similarly, the server uses the subroutine call instruction of its architecture with the target address of the desired routine. The CLASP kernel uses the virtual memory system to detect the transfer to routines that execute on the server. The special instructions used to transfer control in RPC systems are replaced by functions in the CLASP kernel.

Because RPC systems use special calling sequences to invoke routines on remote processors, they do not provide transparent means to pass local and remote procedures as formal parameters to subroutines. CLASP allows applications to pass both local and remote functions as arguments to subroutines. The called subroutine does not require any special processing to handle both local and remote functions; it uses the same instruction sequence to invoke both function types.

RPC systems have several advantages over CLASP. Because the RPC client and server execute in separate address spaces, one half can be changed without affecting the other. Thus, new RPC servers can be installed without changes to existing RPC clients.[8] Program changes in a CLASP-based system require that the program be relinked to incorporate the new code.

RPC servers can be more secure than a CLASP program. Because the RPC server executes in its own address space, all interaction is through the RPC call interface; clients can not modify or destroy data stored in the RPC server. CLASP programs, because they execute in a single address space, do not have this separation.

RPC systems operate between systems with different internal data representations. Because all information passed between systems is contained in the paramters and results of subroutine calls, appropriate typing and conversion operations can be applied when the data is transferred between systems. CAPCs, because they transfer data between systems as pages

[8] Some RPC implementations might require that no client of that RPC server be active when the new version is installed. Others allow active clients to continue with the old server while new clients are connected to the new server.

of uninterpreted data, only operate between systems that share internal data representations.

Cross–Architecture Procedure Calls do a better job of emulating the normal procedure call than the Remote Procedure Call. Because CAPCs provide transparent access to routines on remote processors, they do not require changes to application code. RPCs usually require changes to application code. By eliminating the need to change application code, the CAPC eliminates extra programming costs.

CHAPTER 4.

IMPLEMENTATION OVERVIEW

This section of the thesis describes a prototype CLASP system. This implementation is based on Release 3.0 of the SUN UNIX Operating System [21]. The implementation uses SUN–3 workstations with Motorola 68020 processors. In the next sections, we describe the three components of our prototype: the CLASP loader, the CLASP daemon, and the operating system kernel modifications. The CLASP loader assembles object files for several processor architectures into a single executable file. The CLASP daemon cooperates with the modified kernel to instantiate server processes and record CLASP statistics. The kernel modifications detect and perform CAPC calls and returns; they also manage page residency for each CLASP process's virtual address space.

4.1. The CLASP Loader

Our new loader is a modified version of the standard UNIX loader. The standard UNIX loader processes object files for a single architecture to generate an executable file. The new loader processes object files for several processor architectures to generate a multi–architecture executable file. This multi–architecture executable file differs from a single–architecture executable because it has several text segments. Standard UNIX executable files have a single text segment. Each of the CLASP text segments contains instructions for a different processor architecture.

The multi-architecture executable file is similar to a standard UNIX executable file. The text segment of the multi-architecture file is divided into several text subsegments; each of these subsegments contains instructions for a different processor architecture. The file header of a multi-architecture executable file contains extra information describing how the text segment is partitioned — which sections of the address space correspond to a particular processor architecture.

4.1.1. Text Partitioning

As the loader processes each object file, it determines the processor architecture from header information stored in that file. The loader places instructions for the each architecture in a contiguous segment of virtual memory. Segments begin on page boundaries; no single page contains information for more than one segment. This allows the CLASP kernel to detect non-local instruction references using the hardware supported page-level protections.

The loader provides the kernel with a table that describes the multiple text subsegments. This table specifies the architecture for each subsegment, and its location and extant in the address space. Figure 4.1 illustrates the C structure `clasphdr` that defines this information. This structure is stored at the beginning of the executable file, just after the UNIX *a.out* header information.

4.1.2. Replicated Code

Short routines — like *sqrt*() — are executed most effectively on the local processor. The network overhead to make a remote call and return makes it much more expensive to execute

```
#define MAXCLASPSEGS    4                         /* max # architectures */

struct claspseg
{
    unsigned short  cs_arch;                      /* architecture */
    unsigned long   cs_relocation;                /* address shifts */
    unsigned long   cs_first;                     /* addr in 1st page */
    unsigned long   cs_last;                      /* addr in last page */
};

struct clasphdr
{
    long    c_nsegs;                              /* number of segments */
    struct claspseg c_segment[MAXCLASPSEGS];
};
```

Figure 4.1
Clasp Header Structure

such routines on the remote processor. The network overhead overwhelms any speedup gained by executing on the remote processor. If the routine is called infrequently, this overhead is not a significant fraction of the total running time. However if the routine is called for each element of a large array, the network overhead is unacceptable. In some cases, both client and server make many calls to the same subroutine.

These cases appear to require substantial overhead — it seems that one of the processors must use remote operations to invoke the shared subroutine. A modified CLASP loader can eliminate this overhead by allowing multiple instances of the same function.[1] Each instance of the function executes on a different architecture. If both implementations are compiled from the same source, their execution will generate the same outputs. This follows

[1] At some point, the programmer still must decide which routines should be replicated. The loader does not make this decision.

from the properties discussed in Section 3.2.

The loader symbol table allows multiple instances of symbols that reference addresses in the text segment. When the relocation information uses a text symbol, the loader attempts to use an instance of the label defined for the current architecture. If no instance of the routine exists for the local architecture, the loader uses the instance for the remote architecture.

The loader's attempts to use local instances of replicated routines can fail when procedures are passed as formal arguments to routines. The UNIX *qsort*(3) routine expects an array of elements, the size of the array, and a pointer to a comparison routine. Qsort invokes the comparison routine to determine if the array elements are in order. The call to qsort will be resolved with a local instance of the comparison routine (assuming it is replicated on both processors). If qsort executes on the remote processor, each call to the comparison routine will cause a CAPC call back to the client processor. At this time, we have no general solution to this problem.

4.2. Operating System Kernel Modifications

This section describes how the CLASP kernel recognizes multi–architecture executable files, performs CAPC calls and returns, and transfers pages of the virtual address space between client and server systems.

Although the described implementation is based on the SUN Unix 3.0 kernel, many details should be portable to other variations of the UNIX system, including the AT&T System V standard [21,54].

4.2.1. Recognizing CLASP Executable Files

The UNIX kernel determines the type of an executable file from the *a.out* header. This header describes the architecture appropriate for the executable image. It also contains a *magic number* that describes how the image should be loaded into the virtual address space. Some magic numbers (e.g., *OMAGIC*) specify a process that runs with an impure text segment; these processes can overwrite their instruction space. The *ZMAGIC* magic number specifies a process that is demand–paged from the executable file and shares its (read–only) text segment with other processes executing the same image. Our prototype defines a new magic number — *CMAGIC* — that specifies a ***demand–paged multi–architecture*** executable file. The CLASP loader uses the CMAGIC value in the a.out headers for executable images that it generates.

The kernel loads executable images as part of the *exec*(2) system call. Our new kernel includes the CMAGIC value in the list of allowed magic numbers.

As part of the *exec*(2) call, the kernel builds a new address space from the executable image, replaces the existing address space, and starts the user process at a specified entry point. For CMAGIC files, the kernel instantiates a server process before beginning execution of the user process.

4.2.2. Per–Process Structures

The UNIX kernel maintains a u and proc structure for each process in the system.[2]

[2] These structures define the *user* and *process* information for each active process in a UNIX system. Additional information on these structures can be found in the literature [62].

These structures contain file descriptors, scheduling information, address space boundaries, and other state data for that process. CLASP processes require additional kernel information: the connection to the server processor, text subsegment boundaries, and other information. The `u` structure for each process now contains an extra field pointing to a `claspData` structure for that process. This structure is allocated from the kernel's buffer pool dynamically when a CMAGIC process starts. When the process terminates, the space is returned to the buffer pool. The space requirements for non–CLASP processes are not increased.[3] Figure 4.2 shows the C definition of the `claspData` structure.

Our prototype also stores a copy of the `clasphdr` structure from the executable file in the `u` structure. It should be stored in the `claspData` structure.

4.2.3. Virtual Memory

Our prototype required two modifications to the virtual memory system. The first modification allows the CLASP kernel routines to protect instructions for non–local architectures so they can not be executed on the local processor. The second set of modifications provides the paging functions to share the virtual address space between the client and server processes.

[3] The `u` structure is an integral number of pages. The `claspData` pointer uses space that is already allocated to that structure but is otherwise unused.

```
struct claspData
{
    struct wire_t   cd_wire;                          /* for network I/O */
    struct claspNetwork cd_cn;                        /* the message */
    struct claspNetwork cd_cn2;                       /* 2-message ops */
    int     cd_havedata;                              /* cd_cn is full */
    int     cd_amsegment[MAXCLASPSEGS];               /* segs i do */
    long    cd_textbase;                              /* where they start */
    long    cd_database;
    size_t cd_dataadjust;                             /* used on server */

    long    cd_didshrink;                             /* if brk() shrunk */
    size_t cd_tshrink;                                /* should never shrink */
    size_t cd_dshrink;                                /* smallest dsize */
    size_t cd_sshrink;                                /* smallest ssize */
    long    cd_flags;                                 /* state flags */
#define         CD_READY        0x1                   /* in use */
#define         CD_HAVEWIRE     0x2                   /* got one */
#define         CD_CONNECTED    0x4                   /* and connected */

    /*
     * instrumentation....
     */
    long    cd_pageouts;                              /* pages sent */
    long    cd_pageins;                               /* pages yanked */
    long    cd_pagesize;                              /* across wire */
    long    cd_localcalls;                            /* i handle */
    long    cd_localrets;
    long    cd_netcalls;                              /* to peer */
    long    cd_netrets;

};
```

Figure 4.2
Dynamically Allocated structure for each CLASP process

4.2.3.1. Page Protections

The UNIX kernel already provides internal functions that protect individual pages of the address space. Existing kernel routines use this function to protect the text segment against write access. A new function uses the single–page protection routine to protect a series of pages. The new function expects an address range and a protection and invokes the single–page function on each page in the specified range. When starting a CLASP process, the kernel uses this routine to protect the sections of the text segment that contain instructions for the non–local architecture.

4.2.3.2. Network Page Faults

The second set of modifications to the UNIX kernel implement address space sharing between the client and server processors. During program execution, the pages of the virtual address space move between the client and server processor. When a page is resident on the server processor, the client processor can not access that page. Attempts to access the page generate a memory fault and the kernel must retrieve the page from the server processor. In this respect, the kernel must handle server–resident pages in the same fashion as pages stored on the swap device. The mechanism used to retrieve server–resident pages differs from that used to retrieve pages stored on the swap device.

When a page is non–resident, the corresponding page table entry (*PTE*) contains information describing its location on the swap device. Some pages, which have never been resident, are not stored on the swap device. Instead, the first access to these pages causes the kernel to allocate a page filled with zeroes. These pages are called *fill–on–demand* pages. There are several kinds of fill–on–demand pages: fill with zero, fill from an arbitrary file, and fill from the executable image. The system pagein routine allocates and validates a physical page before invoking a fill–on–demand handler. The handler proceeds by loading the page with the appropriate contents.

Server-resident pages are marked as a new type of fill–on–demand page. These pages are marked *CLASP fill-on-demand.* The pagefault handler's table of fill–on–demand handlers contains a new entry for the CLASP fill–on–demand page type. When processing a CLASP fill–on–demand fault, the pagein routine calls the new *clasp_pagein()* routine with the virtual address of the page and the length of that page.

The *clasp_pagein*() routine issues a request to the server for the appropriate section of the address space. *Clasp_pagein*() uses the communications descriptor stored in the `claspData` structure to communicate with the server process. The server process replies with a message describing the page and the contents of the page.[4] The client stores the retrieved page at the appropriate virtual address, marks the PTE as valid, and returns control to the system pagein handler. After sending a page to the client, the server invalidates its copy of the page: it frees the physical page frame and marks the appropriate PTE as CLASP fill–on–demand.

When a page is retrieved from the server, the client marks the page modified, or *dirty.* This happens even though the client has not modified the page. Otherwise, the local pageout daemon might discard the copy of the page on the assumption that has not been modified since it was last written to the swap device. This is more straightforward than determining whether the remote processor modified the page. It does not require extra information in the PTE to store an extra *modified since retrieved from remote* bit. The cost for this decision is that a page may occasionally be written to the backing store twice. However, the extra disk transfer occurrs asycnchronously, it only affects the pages that aren't modified again by the

[4] While the control thread is on the client, the server process sits in the *clasp_rcv*() routine. For network paging operations, *clasp_rcv*() invokes the proper CLASP paging routines. When a control transfer message arrives, the *clasp_rcv*() routine returns to its caller — the *clasp_fault*() routine.

local processor, and should only occur for pages that fall out of the working set.

4.2.4. Cross–Architecture Calls

Applications make CAPC calls using the subroutine call mechanism of the local architecture. For CAPCs, the target address is a section of the virtual address space protected against execution. When the processor attempts to fetch the first instruction of the called subroutine, the virtual memory hardware generates a protection violation signal. At this time, the kernel *trap*() routine is called to handle the fault.

The trap routine calls the new *clasp_fault*() routine to handle protection violations. *Clasp_fault*() determines whether the fault was caused by a CAPC call, a CAPC return, or is a stray memory reference. For stray memory references, *clasp_fault*() returns an indication to the *trap*() routine to generate a segmentation violation signal for the user process. *Clasp_fault*() decodes the instruction that generated the fault to determine whether it is a CAPC call or return. For CAPC calls and returns, the *clasp_fault*() routine packages the call information and transfers control to the remote processor.

For CAPC calls, *clasp_fault*() determines the argument vector, the target address and the return address. The target address — the address of the called subroutine — is the address that caused the MMU to generate the fault. Machine–specific code examines the stack and also examines the intructions following the call instruction to determine the length and location of the argument vector. *Clasp_fault*() stores this information in a `claspPacket` structure and sends it to the remote processor using the network descriptor stored in the `claspData` structure.

Clasp_fault() uses the *clasp_xmit*() routine to send the control transfer message to the remote processor. After sending the message, *clasp_fault*() calls the *clasp_rcv*() routine to receive the message that returns control to the local processor. The thread can come back as a nested CAPC call. It can also return to the client as a CAPC return. *Clasp_rcv*() relinquishes control of the CPU while awaiting messages. The process *sleeps* waiting for data to arrive on the network file descriptor.[5]

While awaiting the control transfer message, the server process is in the *clasp_fault*() routine.[6] When the message arrives, *clasp_fault*() unpacks the message, builds a call frame on the (server) processor, and sets the appropriate values for the program counter, stack pointer, and other registers. For some cases, an argument register can be loaded with an appropriate value pointing to the arguments. In other cases, *clasp_fault*() must make a copy of the argument vector. After building the call frame, *clasp_fault*() returns through the kernel trap handler with an indication that the user process should be resumed with the new register values. At this time, the user program continues execution on the server processor.

4.2.5. Cross–Architecture Returns

On CAPC returns, *clasp_fault*() takes the return values from the appropriate registers and stores them in the fields of the `claspPacket` passed from the client at call time. *Clasp_fault*() then sends this packet back to the client processor. After sending the packet,

[5] The kernel *sleep*() routine causes the current process to await a specific event. The current process relinquishes control of the CPU and waits for the event. When the event occurs, another process will make a call to the kernel *wakeup*() routine, which will move the sleeping process into the ready queue.

[6] It is really in the *clasp_rcv*() routine while waiting for the message. However, as soon as the message arrives, *clasp_rcv*() returns to *clasp_fault*(). *Clasp_rcv*() also handles the server side of network paging requests. Client requests for pages in memory are directed to the *clasp_pageout*() routine from within *clasp_rcv*().

clasp_fault() waits for memory and control transfer requests.

Upon receiving the claspPacket, the client unpacks the return values and loads the appropriate registers for the client architecture. *Clasp_fault*() sets up the appropriate register values for returns in the same fashion it sets up registers for calls.

4.3. The CLASP Daemon

The CLASP daemon, *claspd*, listens on a TCP/IP socket for connections from client processes.[7] When it receives a connection, *claspd* forks a child process to act as server for that client. After spawning the server process, *claspd* awaits further service requests. The client and its server process communicate independently from the CLASP daemon. The child process uses a new CLASP–specific system call to become a server. This system call accepts the file descriptor of the network connection to the client as a parameter.

The *claspd*() system call allocates a claspData structure for the current process. The descriptor for the network connection is stored in this structure. *Claspd* then collects information across the network from the client process. This information includes the text segment partitioning and a copy of the text segment. The CLASP kernel code returns to user mode after arranging for an immediate protection trap and setting a flag for *clasp_fault*().

The system trap handler calls *clasp_fault*() when the user code generates the protection violation. *Clasp_fault*() examines the flag set by the *claspd*() system call to see that this is a *server initialization* fault. Instead of sending a message to the client process, *clasp_fault*() waits for a control transfer message from the client.

[7] TCP/IP is a stream–oriented protocol that provides in–order, guaranteed delivery communications between two endpoints [73].

4.4. CLASP Algorithms and Protocols

The next several sections describe the protocols for choosing a compute server, establishing contact with a compute server, transferring control between servers (the client is also a server), and transferring memory between servers. The different structures for control and memory transfer are part of a single network structure. The examples in this chapter include only the relevant sections of the network structure.

4.4.1. Server Availability and Selection

A new kernel table is used to select a server processor. New system calls allow *claspd* to clear, replace entries, and append entries to this table. At startup time, *claspd* reads the file */usr/local/etc/claspd.config* for profiling, logging, and server address information. Appendix D describes the syntax of the claspd.config file and includes an example configuration file.

After determining an address for the server, the CLASP kernel routines try to establish a connection to that address. If the connection fails, the client process is aborted.

4.4.2. Initiating a Dialogue with a Compute Server

CLASP clients establish connections to the required server processors when the client process is instantiated. In our implementation, the operating system establishes these connections as part of the *exec*(2) system call. Once a server process has been started, the client process initializes the server by sending a description of the address space. This description includes the address space bounds and the text subsegment locations, sizes, and architectures. After this information is transmitted, the client process sends a copy of the text segment across the network to the server.[8]

Our prototype does not try to connect to alternate servers if the first server does not respond. Our implementation does not provide a protocol for a server to deny service on a selective basis.

4.4.3. Calling Procedures on a Compute Server

Clasp_fault() determines the target address for remote calls, the location and length of the argument vector, the return address, and the stack bounds. This information is stored in a `claspPacket` structure. Figure 4.4 shows the C declaration for this structure.

Clasp_fault() stores a copy of the `claspPacket` structure on the user stack. It builds a message to the server that contains the address of the `claspPacket` on the stack and a copy

```
struct initiate_data
{
    struct clasphdr cnx_claspHdr;                       /* from a.out */
    long    cnx_caller_segments[MAXCLASPSEGS + 1];      /* client is */
    long    cnx_callee_segments[MAXCLASPSEGS + 1];      /* server is */
    long    cnx_textbase;                               /* segment bases */
    long    cnx_database;
};
```

Figure 4.3
Data Transferred to Start A Server
[part of the claspNetwork structure]

[8] Pushing a copy of the entire text segment across the network can be expensive. Some processes have as much as a megabyte of instruction space. A future implementation might pass file handles (like the NFS rnode) so the server can demand page portions of the address space from the file on the client processor [15].

```
struct claspPacket
{
    long    cp_action;                          /* call, return, etc */
    /*
     * values used in a call
     */
    caddr_t cp_subroutine;                      /* address to call */
    caddr_t cp_sp;                              /* where server can start */
    caddr_t cp_arglist;                         /* base of arg vector */
    long    cp_arglen;                          /* length of arg vector */
    /*
     * values used in a return
     */
    caddr_t cp_return;                          /* return address */
    caddr_t cp_usp;                             /* user SP after return */
    unsigned long   cp_r0;                      /* return 0 */
    unsigned long   cp_r1;                      /* return 1 */
    /*
     * server end of a call uses this for temp storage
     */
    struct claspPacket *cp_lastp;               /* for nested cross-calls */
};
```

Figure 4.4
CAPC Information Packet

of the claspPacket. The controlxfer structure, depicted in figure 4.5, is part of the larger claspNetwork structure. Other fields in the claspNetwork structure contain the current address space boundaries; these fields are loaded before the message is sent to the remote processor.

4.4.4. Returning from Procedures on a Compute Server

For returns, *clasp_fault*() uses the existing claspPacket from the CAPC call. The cp_action field is changed from call to return, and the return value fields are loaded with

```
/*
 *      The claspNetwork combined structure contains fields describing
 *      the extent of the address space.
 */

struct controlxfer {
    long cnx_ctlxfertype;                   /* call or return */
    struct claspPacket *cnx_cpp;            /* pointer to CAPC frame */
    struct claspPacket cnx_fastcapc;        /* copy of CAPC frame */
};
```

Figure 4.5
CAPC Control Transfer Information

values from the appropriate processor registers.

Clasp_fault() then follows the same steps to send this claspPacket back to the client as it would to send a call message to the client: address space bounds are loaded into the claspNetwork structure and the message is written on the network descriptor in the claspData structure.

4.4.5. Memory Transfers Between Clients and Servers

The CLASP prototype memory system uses a simple model to maintain coherency in the virtual address space: each page of the address space resides on exactly one host. If one CPU needs a page that resides on another host, the page is demand–paged from the remote processor to the local processor. The local processor — the one that wants the page — is the client. The non–local processor that currently holds the page is the server. A processor acts as both client and server at various times through the lifetime of an application process.

When the CPU attempts to access a page that is not resident in main memory, the MMU generates a page fault. Non–resident pages can be retrieved from several different locations. Some pages are retrieved from the local backing store (e.g., the swap space of the local processor). Other pages reside on remote processors and must be retrieved across the network. In both cases, the page table entry describes where the page is stored and how to retrieve the page.

The client CLASP kernel marks pages resident on the server as *fill–on–demand–clasp*. This allowed us to implement the shared virtual address space without changing the width or bit assignments of fields in the page tables. The kernel pagein routines treat fill–on–demand–clasp similarly to fill–on–demand-zero. Instead of zeroing a page and validating it for the user, page faults on fill–on–demand–clasp pages cause a call to a CLASP–specific pagein routine which retrieves the page from the remote processor and places it in the page frame allocated by the normal pagein routine. The changes to the normal pagein routine are limited to an additional case in the switch statement that handles fill–on–demand pages.

The client sends MEMGET requests to the server to retrieve pages that reside on the server. These MEMGET packets contain the above `memxfer` structures to describe the pages requested. Servers reply with a MEMPUT packet that describes the pages being returned and follow that packet with the data of the page. If the client and server have different page sizes, the MEMPUT packet might describe a different (e.g., larger) block of memory. A client with 512 byte pages making a request to a server with 1024 byte pages would receive two 512 byte pages.[9]

[9] These differences should be resolved when a CLASP process begins execution. The client and server should agree on a network page size that meets their individual requirements for local pages. All network paging operations should be done in units of this agreed page size.

```
struct memxfer
{
    caddr_t cnx_base;                       /* base in vaddr */
    caddr_t cnx_length;                     /* byte count */
};
```

Figure 4.6
Information Sent for Memory Transfers

Once pages are retrieved from the server, the client marks them *dirty*, or modified, to inform the pageout daemon that these pages have been changed since the last time they were written to the local swapping device. This assumes that the server modified the page. While it might not always be true, the alternative was to add several additional dirty bits to the page table entry — one for each backing store that might hold the page.

This simple memory model made our prototype easy to implement. However, the model limits performance by keeping only one copy of any page in the address space. If a processor has a local copy of a given page, the overhead of a page fault across the network can be avoided. One problem with replicating pages on each processor is maintaining the coherency of the virtual address space. The replication of read–only data is a simple and obvious way to improve performance.[10] We describe some other work that supports multiple instances of pages in the address space in section 6.2, under *Further Performance Optimizations*.

[10] Our prototype keeps copies of the text (code) segment on both machines. The CLASP text–segment is filled with read–only data (e.g., the instructions).

CHAPTER 5.

PERFORMANCE OF THE IMPLEMENTATION

CAPC allows an application program to be partitioned so that some routines execute on a faster server processor to decrease the running time of the program. Partitioning an applications program always introduces some overhead: CAPC subroutine calls and returns are slower than local subroutine calls and returns. Data residency also affects the partitioning's overhead. Some pages are accessed by the routines on the client; others are accessed only by server routines. Some pages are accessed by both client and server routines. Pages accessed by both processors must be moved to the appropriate processor when needed.

To overcome this overhead, the server must be faster than the client processor. The breakeven point can be derived from the paging behavior, CAPC calling patterns, and speed differential of the two processors. This breakeven point is different for each program and can vary within a single program depending on how it is partitioned.

In this chapter, we discuss types of algorithms that will perform well under the CLASP system. We also discuss an existing model that characterizes paging behavior [32,33]. We present the results of benchmarks to determine the costs of our system. These costs include remote call overhead and network paging overhead. In appendix B, we show the results of several benchmark programs under the CLASP system. Section 5.3 compares our empirical results with those predicted by section 5.1. The chapter closes with a discussion of several mechanisms for partitioning applications between client and server systems.

5.1. Theoretical Performance Expectations

To determine whether part of a program should be moved to a server processor, an appropriate question to ask is: ***is the execution speedup greater than the communications costs?*** If data transmission costs are greater than the possible speedup, the problem should not be moved to the server processor.

Several factors affect the performance of a partitioned application. Programs where the execution costs grow faster than the communications costs to move data between client and server quickly overcome the communications overhead. Algorithms that access portions of a data structure, such as tree searches, are another class of algorithms that can yield improved performance when partitioned between processors.

5.1.1. Algorithms Appropriate for the CLASP architecture

Algorithms for solving linear systems are a good example of problems where improved execution time on the server recovers the communications time between the client and server. Linear systems of order n comprise an nxn matrix. The cost to move this problem to a remote processor across a network is $O(n*n)$. The time to factor this matrix is $O(n*n*n)$. The breakeven point occurs when the speedup on the remote processor matches the communications cost to move the data to the remote processor. If A(n) is the savings in execution and T(n) is the communications cost, our breakeven point occurs when we satisfy the following equation.

$$T(n)=A(n)$$

C-3

For an applications using gaussian elimination to factor matrices of order n, the appropriate equation is:[1]

$$K_1 n^* n = K_2 n^* n^* n$$

The exact value of n that satisfies this equation depends on the constants. These constants are determined by the network speed, client processor speed, and server processor speed.

To determine if there exists an n where the problem should be moved to the processor, we examine the inequality:

$$\lim_{n \to \infty} \frac{T(n)}{A(n)}$$

If this limit is less than one, there will be some problem size n that executes faster in a partitioned environment.

We do not want to give the impression that only higher order algorithms are applicable to our architecture. A number of data structures have search times smaller than $O(n)$. Trees searches execute in time $O(log\ n)$. These searches do not require the entire data structure to be resident on the local processor. The $O(log\ n)$ probes of a tree search will move at most $O(log\ n)$ pages from the remote processor. Additional searches, which often probe the same initial nodes of the tree, will generate fewer page faults. Insertions, balancing, and other tree operations can often execute with only portions of the data structure resident.

Other data structures that require less than $O(n)$ time to manipulate are hash tables, queues, lists, and heaps. For these data structures, the communications costs are a function

[1] For this example, we have dropped the lower order terms from the algorithm costs.

of the probes into the structure.

5.1.2. Localized Data

Some data structures are accessed only by several routines. If all of these routines are implemented on the same processor, there are never any communications costs to access that data.[2]

In such cases, the costs to move this portion of an application to the program are related only to the frequency and duration of the calls to those subroutines. If the subroutines execute for more than our CAPC call/return overhead, we expect improved performance by moving them to a faster processor. The exact client/server speed ratio needed to compensate for the CAPC call/return overhead depends on the time for the subroutine call on the client.

5.1.3. Paging Patterns

In 1968, Peter Denning introduced the *working set* model to help manage page traffic in virtual memory systems. The working set model uses recent page access history to predict the short term memory needs of a program. The objective is to keep a *working set* of pages resident in memory and increase the average *instruction burst* between page faults. The working set model is based on the observation that programs show localized access patterns. Working sets exhibit *slow drift* behavior; the working set changes gradually over time [31].

[2] This ignores the boundary condition when we start the process and the entire address space is resident on the client or the client's swapping device.

By 1974, Denning and others determined that the slow drift concept was wrong [32]. While programs did have phases which showed slow drift behavior, these programs also displayed disruptive transitions between phases. Most phases used almost completely different sets of pages, or *locality sets*[24,25,33,51,53,63]. Kahn found the following about phases and transitions between phases:[53]

- Phases covered 98 percent of the virtual time.
- 40 percent to 50 percent of the working set page faults occurred in transition periods. Thus, about half of the paging occurred in 2 percent of the virtual time.
- The same phases were observed by the working set policy over wide ranges of its control parameters.
- Fault rates in transitions were 100 to 1000 times higher than fault rates in phases.

Other observations indicate that approximately 90 percent of the virtual time is spent in long (at least 100,000 memory references) phases. These long phases account for only 10% of the recognized phases, the other phases are fleeting and embedded within transition periods.

To model phases and the transitions between phases, Denning built a macro–model using semi–Markov chains. The states in this model correspond to phases and their locality sets. The holding time of each state corresponds to the phase length for that locality set. Within each state, Denning used existing micro–models to generate access patterns across the pages in the locality set. Denning found that this macro–model followed the behavior of real programs better than the existing models.

5.1.4. Performance Expectations

We expect that the CLASP architecture will work well for a number of applications. These applications will have one or more of the following properties:

- algorithms where the execution costs grow faster than the communications costs. An example is the solution of **Ax** = **b**, which requires O(n*n) communications and O(n*n*n) execution time.
- algorithms which exhibit high degrees of locality.
- access methods, such as hashing and tree searching, which move small parts of a larger data structure.
- subroutines that encapsulate access to data structures. These data structures will not move between processors, so the communications costs are only the procedure call overhead.

Where communications and execution costs are of different magnitudes, the advantages are apparent. Where both costs are of the same magnitude, the coefficients become more important. In both cases, the exact breakeven point depends on the particular application and its calling patterns. In some cases, demand–paging saves two network faults — such as when the results of one remote operation are passed directly to another remote operation.

5.2. Empirical Results

We ran a series of benchmark programs with our CLASP kernel to obtain a measure of its performance. Some benchmarks provided us with the overhead of the capc mechanism and paging costs between client and server. Several benchmark programs, acquired from other sources, were used to generate information on the frequency of paging traffic between client and server systems. In this section, we discuss the benchmarks used to determine the CAPC call/return overhead and the network paging costs. Appendix B contains performance data for other benchmarks and looks at timing, speedup potential, and paging behavior of those benchmarks. The next section (*5.3*) compares the observed paging patterns with the behavior we predicted in section 5.1.

Remote operations — both calls and data accesses — are almost always significantly more expensive than local calls. The network overhead accounts for most of this difference. We can divide the network overhead into two pieces: latency and bandwidth. For smaller messages (such as control transfer packets), network latency dominates the overhead. Other research provides insights towards developing low–latency, high–bandwidth networks between processors [50,78].

5.2.1. Remote Call and Paging Costs

Figure 5.1 contains the times for CAPC calls and returns in our prototype. The CAPC overhead number represents the time for the client to invoke a subroutine on the server with no arguments and for that subroutine to return. The number is an average across 10,000 invocations of the subroutine. Timings of individual calls — to obtain minimum, maximum, and standard deviation figures — was not possible with our hardware. The clock on our systems ticks at 50 Hz, the smallest interval we can measure is 20 milliseconds.

Figure 5.1 also contains the time for network paging. We used two different programs to generate this data. The first page fault program is based on the CAPC call overhead program. In this version, the calling routine accesses a global variable once during each iteration of the loop. The server routine accesses this same variable once during each call. This causes 2 page faults for each CAPC call/return pair. We determined paging costs by subtracting the known CAPC call overhead. From this program, we can determine the time for a pair of page faults.

The second program alternates calls to subroutines on the client and server that step through a large array. We ran this test with arrays ranging from 100 through 400 pages — 8192 Kbytes through 3276 Kbytes. Arrays larger than approximately 2200 Kbytes filled the available physical memory and introduced other factors into the times; the desired page would not be resident in main memory and had to be retrieved from the swapping device. The numbers in figure 5.1 reflect the times for 2000 Kbyte arrays. Because this program generates long strings of page faults in one direction, we can time those strings to determine the times to send or receive pages. This allows us to break down the round-trip costs obtained in the first paging benchmark.

These benchmarks were run using the loopback interface to the same processor. These timings show that the process spends large amounts of time in the system kernel. We believe that most of this time is TCP/IP protocol overhead. With 53 milliseconds per page fault, our prototype kernel can process 18.8 pages or 154,000 bytes per second. Additional benchmarks showed that a TCP socket could only move 297,000 bytes per second on our system. Our prototype provides more than half of the throughput with the current network protocols. Synchronous page requests and page transfers account for the lost page bandwidth.

Figure 5.2 shows the times for empty procedure calls using CAPC, Sun RPC (both UDP and TCP transport mechanisms), and Xerox Courier. In all but one case, the times are averages across 10,000 calls. The times for Courier with the standard kernel for 1,000 calls. The Courier times for the standard kernel are more than an order of magnitude slower than the other mechanisms. This is caused by the SUN TCP implementation. Instead of flooding the network with small TCP packets, the SUN TCP code delays small packets so it can combine them into a larger packet. If no further data arrives, a timer signals the TCP code to send

CAPC Test Case	Client system time	Server system time	Wall time
CAPC Call Overhead	4.46	4.45	8.98
CAPC + 2 page transfers	56.74	56.70	115.44
2 page transfers	52.06		106.42
1 Page Transfer	26.03		53.21
CAPC + 250 page xfers			
avg/page	26.34	26.86	53.42
client->server avg/page	30.36		53.52
server->client avg/page	22.28		52.92

Figure 5.1
CAPC Overheads
[all times in milliseconds]

the small packet. From these times, we can see that the timer fires every 200 milliseconds. For these tests, the Courier code sends messages that fall below this threshold. We modified the kernel to lower this threshold and re-executed the Courier benchmarks.[3] This small packet threshold does not affect the timings for the other benchmarks because they send larger packets between client and server.

For small argument vectors, the CAPC call packet contains a copy of the argument vector. Larger argument vectors are passed by pointer; the server will demand page the argument vector across the network. Our prototype sends up to 64 bytes of arguments in the CAPC call packet. For these small argument vectors, a CAPC costs approximately 9 milliseconds. For larger argument lists, the call time is 9 milliseconds plus the time to move the appropriate stack pages to the server. Because these stack pages are required on the client after the called subroutine returns, they must be paged back from the server routine. In

[3] Other TCP implementations we have seen do not have this small packet threshold. For example, the 4.2 BSD kernel does not hold these small messages. Also, it is worth noting that most Courier calls will be larger than the 10 byte small-message threshold. Return messages, if they are simple integer values, fall below the threshold.

Mechanism	User Time	System Time	Wall Time
CAPC	0.0	4.46	8.98
Sun RPC (udp)	0.432	3.076	7.942
Sun RPC (tcp)	0.846	3.118	8.014
Courier (stock kernel)	0.0	0.020	399.960
Courier (modified kernel)	0.48	5.26	11.78

Figure 5.2
Empty Call Costs
[all times in milliseconds]

most cases, argument vectors larger than 64 bytes will take approximately 115 milliseconds. If the argument list crosses page boundaries (the sun–3 hardware uses 8 kbyte pages), the costs go up by another 2 page faults — another 100 milliseconds. Figure 5.3 shows the average costs for CAPC calls with argument vectors ranging from 0 to 1024 bytes. These times are to set up, execute, and return from the remote subroutine.

Since most procedure calls have small argument lists it is sensible to spend our efforts making the most frequent case execute quickly. Code analysis done for RISC machines has shown that, in a UNIX environment, many procedure calls have fewer than 6 arguments. These procedures often account for more than one–half of the dynamically executed procedure calls [35,70]. Our prototype, which provides 64 bytes of ***fast arguments***, handles 16 4–byte arguments before falling back to the slower call mechanism. Therefore, our optimized CAPC calls for small argument vectors should handle most procedure calls. Larger argument lists are processed more slowly, but they make up a small percentage of the subroutine calls.

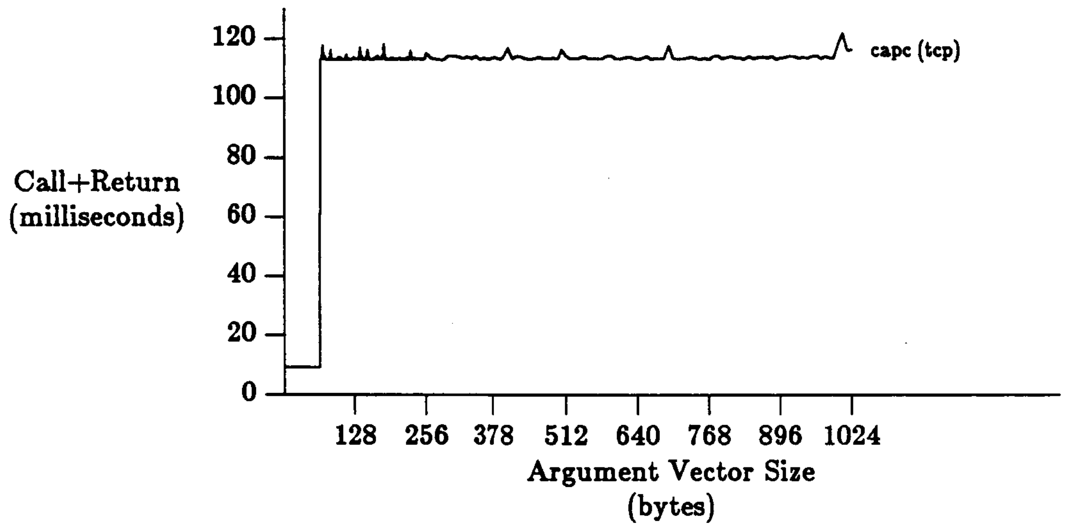

Figure 5.3
CAPC Overheads for Different Argument Lengths

5.3. Comparing the Facts to the Theory

The LINPACK benchmark described in Appendix B showed very good performance in our system. This benchmark moves data of size $O(n*n)$ from client to server and performs $O(n*n*n)$ operations on that data. The breakeven point for this benchmark came for systems of size 59. For a problem of this size, the server had to execute at 17 times the speed of the client processor to compensate for the network overhead. For matrices of order 81, the server had to be only twice as fast as the client to compensate for the network overhead.

This benchmark used matrices dimensioned at for 200x200 systems. This over-allocation generated extra page traffic by removing some of the locality within the array. Another version of the benchmark, using matrices dimensioned to the exact size of the system, shows a better breakeven point. The speed differential can compensate for the overhead on a 42x42 system. A server executing twice as fast as the client breaks even on a 54x54 system.

A second benchmark, the *compress* utility did not show an improvement when partitioned between client and server [87]. This program operates as a filter on its input data. For the test we ran, the network overhead was larger than the total execution time for a single architecture version of the program.

5.4. Considerations For Partitioning Applications

Two factors affect the placement of a routine: the cost to execute a CAPC call and the cost to demand page the required memory to the remote processor. Small subroutines often do not execute enough instructions for the client/server speed differential to overcome the CAPC overhead. Other subroutines may execute enough instructions to make up the CAPC overhead, but their data access patterns may cause an excessive number of page faults.

In terms of Dennings model, we want to partition our program so that cross-architecture calls have a close correlation with transitions between phases. Short procedure calls may generate extra paging traffic and disrupt the phase/transition page fault patterns.

In this section of the thesis, we discuss how these factors can affect performance. The next section presents an existing algorithm for partitioning applications between client and server processors.

5.4.1. Frequency and Duration of Calls

In our implementation, a simple CAPC call and return costs approximately 9 milliseconds. Local subroutine calls, using only several microseconds, can be considered free when compared to 9 milliseconds. If the routine executes locally in less than 9 milliseconds, a remote call will always be slower, regardless of the speed of the remote CPU. If the local time is greater than 9 milliseconds, the breakeven point depends on the relative speeds of the processors (and the data residency).

Ignoring data residency, a subroutine that executes in 18 millisecond on the workstation can be moved to a server that executes twice as fast as the workstation. The server will execute the subroutine in 9 milliseconds, plus the 9 millisecond CAPC overhead, and achieve the same total time as the workstation invocation.

In practice, data residency affects the breakeven point by introducing paging overhead to move the data to the remote processor. Appendix B shows several program examples, how they perform in uniprocessor mode and under CAPC, and what speed differentials are required to break even for different problem sizes.

The frequency of calls to a subroutine affects its placement. If a subroutine is called only a few times during the execution of a program, the overhead of a CAPC call has little impact on the total running time of the program. An extra several hundred milliseconds has little affect when a program executes for minutes or hours. However when calling routines hundreds or thousands of times during the execution of a program, any additional overhead becomes significant. Section 5.5 describes techniques to partition routines between client and server systems to minimize this overhead.

An alternate approach is to replicate these frequently called subroutines on both architectures. For long–executing routines (e.g., factoring large matrices), this is not practical. Replication of long–executing routines is counter–productive; we want these routines to execute on the faster processor. For short subroutines — like sqrt() or strcmp() — replication is important. Without replication, the overhead to invoke remote instances of these routines can overwhelm any performance improvement gained by moving long–executing subroutines to the faster processor.

5.4.2. Data Residency

In our prototype, each page exists on exactly one of the processors. If a pair of procedures on different processors alternately access a page, that page bounces between the processors. We can reduce or eliminate this effect by placing both procedures on the same processor.

Page replication schemes eliminate the problem when neither processor modifies the shared data. Each processor maintains a copy of the page and allows read access by subroutines executing on that processor. If a subroutine attempts to write on a replicated page, the other copy must be updated or invalidated. Again, the time for these operations raises the overhead associated with splitting these routines. But the overhead is often significantly lower than a non–replicated environment.

5.5. Determining Partitionings

Subroutine call frequency and data residency are factors in determining how to partition an application. These factors, by themselves, provide information about costs of a partitioning. However, they do not determine partitionings.

The next two sections discuss several approaches to partitioning applications. The first section discusses several tools that can be used to provide information about call patterns and the duration of procedure calls. The data gathered from these tools can be used to make decisions. It can also be used as input to more formal partitioning schemes. One of these schemes is discussed in section 5.5.2.

5.5.1. Heuristic Partitioning Tools

Profiling tools are useful for identifying where a program spends most of its time. The UNIX utilities *prof* and *gprof* provide different types of profiling. *Prof* provides a summary of the total time spent in a routine and the number of times it was invoked. *Gprof* provides this information and adds the calling patterns between routines. For each routine, gprof reports the calling and called subroutines.

An easy way to determine a partitioning is to execute the program with the profiling tools. From this data, long running routines or sets of routines (e.g. a locality set of subroutines) can be identified and moved to the server.

5.5.2. Theoretical Partitioning Methods

Other researchers have generated algorithms and heuristics for partitioning applications in a distributed environment. These approaches use mathematical tools to determine the partitioning [38,39,79].

In their 1978 paper, Stone and Bokhari use graph theory to determine subroutine placement in distributed systems [79]. Each subroutine in the application is a vertex in their graph. The edges between vertices represent the calling patterns between routines. Each edge is given a weight corresponding to the communications costs if the two vertices (*subroutines*) are on different processors. Two additional vertices, representing the processors, are added to the graph. From each processor vertex, edges are drawn to all subroutine vertices. These edges are assigned weights that correspond to the execution time for that subroutine on the *other* processor. Because the edges represent the costs if two routines are on separate processors, the weight is the execution time for that routine on the remote processor. After generating this graph, they generate a minimal cutset of the that graph. The processor vertices will be in different subgraphs. The two subgraphs contain the subroutines to be loaded on each processor. Stone shows that the minimal cutset generates an optimal placement for the subroutines [79].

Edge weights consider how often one procedure calls another and the data transfered for each call. In our shared memory model, we must also consider routines that interact through shared global variables. This consideration adds edges to the graph for subroutines that do not call each other, but do access the same variables.

We want to bind certain routines to specific processors. For example, most system calls must execute on the client processor — where the appropriate kernel state information resides. We can bind a procedure to a specific processor by adding an infinite weight edge between the vertices for the desired processor and the appropriate subroutine. In the the processor/node context, this indicates that the execution time of the subroutine on the remote processor is infinite, or that it can not execute on the remote processor. In the cutset context, this edge will never be in a minimal cutset. A cutset with this edge would have infinite weight.

These mathematical models can yield optimal or near optimal partitionings for applications. Some of their input data can be gathered from static analysis of programs. The tools discussed in section 5.5.1 can provide additional information for calculating edge weights. Further research into combinations of these tools could provide automatic partitioning schemes that combine all of these tools.

CHAPTER 6.

SUMMARY

This thesis has introduced CLASP, a new software architecture for sharing processor resources. CLASP maps the traditional UNIX process model onto a new foundation. CLASP provides a transparent mechanism for transferring control between processors and allows sections of an application program to execute on the most appropriate architecture. This control transfer is implemented by the *Cross-Architecture Procedure Call* or *CAPC.* CAPCs allow existing applications to be partitioned between processors without making source code changes.

Section 6.1 discusses some additional research suggested by our investigations. Some approaches to reduce the overhead of the CAPC are discussed in section 6.2. The chapter closes with a summary of our results.

6.1. Further Research

There are a number of additional research topics related to CLASP and the CAPC. Some of this research is concerned with improving CAPC performance: using faster network protocols and reducing network paging traffic. These research areas are discussed in section 6.2, *Future Performance Optimizations.*

Other research to add new features to CLASP and to apply existing tools to CLASP systems includes: multiple compute servers, multi-thread computations, multi-architecture debugging, asynchronous traps, I/O operations on servers, process migration, operating system independence, and automatic program partitioning. Each of these is described in more detail in the following sections.

6.1.1. Multiple Servers

Our prototype supports two architecture CLASP programs. The extension to three or more architectures requires additional work in communications between the different processors.

As the number of architectures (and processors) climbs, the replicated address space work of Kai Li becomes a more important factor to reduce the cost of network paging [60,61]. In these situations, a process must determine which CPU has the copy of the page — in addition to moving it to the local host.

6.1.2. Multi–Thread Computations

A number of existing systems provide multiple control threads within the same address space [22,28,30,74]. We would like to see a combination of these systems and our CAPC system. Such a combination would allow programs to use the most appropriate mechanism for performance improvement — parallelism or fast sequential processing — within a single application.

Our current prototype uses the single–thread nature of the Sun UNIX process to streamline some operations. The same agent on the server processes page requests and control transfer requests. A multi–thread implementation would need to partition the address space and control flow operations. Our prototype deferes the propogation of changes in the address space until it passes the control thread to the remote processor. A multiple–thread implementation requires a different address space propogation mechanism.

6.1.3. Debugging

The CLASP system introduces several problems for debugging systems. Symbolic debuggers must now understand the instruction and calling sequences for different processor architectures. The system debugging facilities (e.g., the UNIX *ptrace*(2) system call) must be able to manipulate the control thread of a program when it is on a remote processor.

6.1.4. Asynchronous Traps

The UNIX signal mechanism provides a means to transfer control to a specific routine on the occurrence of specific events. Our prototype does not address the problem of how these signals should be processed when the control thread is on the remote processor.

6.1.5. I/O Operations on the Server

Our prototype performs all system calls, including I/O operations, on the client system. We make this restriction because the existing file descriptors are stored in the kernel on the client system. As long as the correct file descriptors are presented to each system, it should be possible to perform I/O operations on both systems. One approach that supports this operation is to modify the system call templates to select the proper system for the system call. This technique has been used in systems like the Newcastle Connection to redirect system calls to the kernel that holds the appropriate state information [27].

6.1.6. Graceful Process Migration

Because the server kernel only requires information about the address space of a process, it should be able to move all of this state to another processor. There are no file descriptors to move between server processors. Further work with the CLASP system should produce mechanisms that allow servers to re–direct clients to other processors of the same architecture. This can be used to limit the load on a particular server. It might also be used when rebooting a server; existing clients could be moved to other processors.

6.1.7. Operating System Independence

Because the CLASP server maintains only address space information, we should be able to implement servers with an *open systems architecture*. Clients running the UNIX operating system might communicate with servers on other operating systems such as DEC's VMS, CDC's NOS, the Stanford V kernel, CMU's MACH, and other operating systems. The routines that execute on the server do not access system functions, they only use the processor to execute instruction sequences.

6.1.8. Automatic Partitioning

The algorithms described in section 5.5 partition programs to reduce the communications costs and improve the performance of a program. Compilers already generate data dependency information and can generate control flow information. Software generation systems (compilers and tools like the UNIX *make* utility) could use this information to partition applications without user interaction [40].

6.2. Future Performance Optimizations

CLASP systems rely on an underlying network communications system to transfer control between processors and to demand page the virtual address space between processors. Two approaches to reduce the overhead of the network communications are to employ faster, lower–overhead network protocols and to reduce the number of network operations. These two topics are discussed in the following sections.

6.2.1. Network Protocols

Our current CLASP prototype uses the TCP/IP network protocol to communicate between client and server processes. TCP/IP provides a full–duplex, error–free communications channel between two endpoints. The protocol achieves these features at a cost in throughput and latency. However, the other available protocol (UDP/IP) does not provide reliable delivery of messages.

CLASP does not require a stream connection. The CLASP network operations can be mapped directly onto a protocol that provides guaranteed delivery of messages. Such protocols might provide low latency messages, which would improve CLASP network paging times.

6.2.2. Network Paging Performance

Another approach to reducing the network overhead of a CLASP system is to reduce the traffic across the network. In this section, we describe two approaches to reducing the page traffic between CLASP clients and servers.

The UNIX system dynamically extends a program's stack segment to accomodate the calling patterns of that program. The kernel never reduces the size of the stack segment, even if the pages are no longer used by the application program. In our prototype, this generates unnecessary page traffic. If the server extends the stack onto a page that has been used previously by the client, the server retrieves the page through the network. This page, which is about to be overwritten with new data, could have been a *fill-on-demand* page and serviced locally. To eliminate this page traffic, the CLASP system could remove the pages beyond the stack pointer at each CAPC call and return. These pages, located beyond the current stack pointer, should be unused and can be discarded.

Our CLASP prototype maintains a single copy of each page in the virtual address space. This simplifies the page management scheme but increases paging activity. Kai Li and Paul Hudak describe a virtual memory system for loosely-coupled systems [61]. Their system allows multiple instances of each page in the address space. Extra copies of a page are marked *read-only*. Attempts to modify these pages generate traps to the operating system that invalidate the extra copies of the page and proceed with the updates. This approach allows read-only pages (and pages that are read often and written seldom) to be replicated on the appropriate processors.

6.3. Conclusions

In this thesis, we introduce the *Cross-Architecture Procedure Call* or *CAPC*. The CLASP software architecture uses CAPCs to provide access to compute servers. The CAPC is a transparent mechanism to transfer a control thread between two processors. Unlike its predecessor, the Remote Procedure Call, CAPCs allow local and remote procedures to communicate through shared global variables including pointer data types. This allows existing programs, that use these constructs, to be partitioned between client and server processors using CAPCs without any source code changes.

In the first chapter, we propose criteria for our new architecture. These criteria are:

- The user need not restructure or recode his applications.
- The programmer can specify an application's partitioning. Changes to this partitioning do not require changes to the application source code.
- Interactive tasks execute on the workstation. That is, the workstation is not used as a simple terminal to submit jobs to the supercomputer.
- CPU–intensive tasks execute on the supercomputer.
- Optimization techniques, such as vector operation and parallel operations, specific to certain architectures are still useful for code segments executed on those architectures.
- The compilers for each system need not be modified; a modified loader combines the output from the respective compilers into an executable file.
- The operating system resolves issues of control transfer and data transfer between systems.

CLASP meets these criteria. CLASP satisfies the first four criteria because it provides a transparent interface between routines on different processors. Routines on different processors can pass pointer data types and share global variables. This transparency allows users to place routines on the architecture best suited for those routines. CLASP allows each architecture's compilers to apply appropriate optimization techniques to routines that will execute on those processors. Our prototype does not modify existing compilers; it uses a new loader to combine object files for each architecture into a multi–architecture executable file. The CAPC runtime implementation is handled within the operating system. The operating system transparently handles paging traffic between local and remote processors.

Unlike the Remote Procedure Call, the Cross–Architecture Procedure Call does not restrict the interface between procedures. CAPCs model the procedure call interface more completely than RPCs. This feature allows existing applications to be re–compiled for a multi–archiecture environment and yield improved performance without any source code changes.

The Cross–Architecture Procedure Call is an elegant mechanism for accelerating specific portions of applications programs. It extends a simple process model onto a new foundation that provides improved performance without introducing restrictions on calls between procedures, access to global variables, and passing pointers.

APPENDIX A.

ARGUMENTS AND RETURN VALUES

This section describes mechanisms for passing arguments, determining the size of the argument list, and how return values are handled. This is a survey of several architectures and includes: how to determine the argument list location and size when acting as client, how to copy and use the argument list location and size when acting as a server, how to package the return results when acting as a server, and how to interpret the return results when acting as a client.

Each of the systems described in this appendix use the same data representation. They all use the IEEE floating point standard internally. They all store integer values with the same byte ordering.

A.1. Motorola 68000

The Motorola 68000 architecture does not load a register with the address of the argument list as part of the procedure call instruction. Instead, the convention is to place the arguments on the stack. The subroutine call instruction pushes the return address on the stack. At procedure entry, the argument list is located 4 bytes above the current stack pointer. Figure A.1 shows the 68000 stack frame at procedure entry.

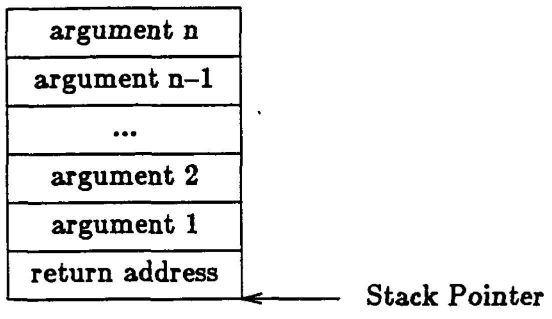

Figure A.1
Motorola 68000 Stack Frame
(at procedure entry)

Compilers for the 68000 architecture generate code that executes a *link* instruction as the first instruction of a subroutine. The *LINK* instruction pushes the contents of a specified register onto the stack and loads that register with the value of the stack pointer. The register modfied by the LINK instruction is then used as a base register, or frame pointer, to access both arguments and local variables for that procedure. The agument vector starts 8 bytes above the value in this register (traditionally register A6). The first 4 bytes above A6 are the previous contents of A6; the next 4 bytes are the return address. The stack after the procedure preamble is shown in figure A.2.

The CLASP client routines for the 68000 architecture determine the length of the argument vector by examining the instruction after the procedure call. Because the hardware does not provide a mechanism for including the length of the argument vector as part of the procedure call instruction, this instruction pops any arguments from the stack. The CLASP kernel decodes this instruction to determine the length of the argument vector. If the next instruction does not pop arguments from the stack, the CLASP kernel assumes the procedure

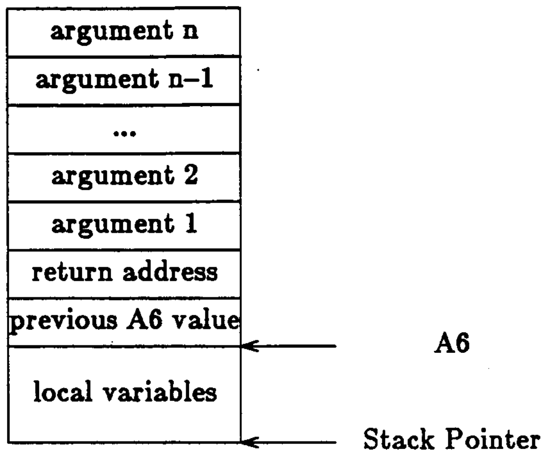

Figure A.2
Motorola 68000 Stack Frame
(after procedure prolog)

has no arguments.

The CLASP server routines for the 68000 architecture copy the argument vector and push the return address to provide the called procedure with a stack that appears to have been generated by the 68000 subroutine call instruction.

Procedures and functions return their results in the D0 and D1 registers. Most functions return their values in the 32-bit D0 register. Floating point results are returned as 64 bit values. For floating point results, both D0 and D1 registers are used.

A.2. Alliant FX Series

The Alliant FX series is multiprocessor system that contains several *computation elements* and *I/O processors.* The I/O processors are members of the Motorola 68000 family.

The computation elements provide a superset of the 68000 instruction set. Both processors use the same stack formats and calling sequences. The Motorola 68000 section of this appendix contains more information about this stack format, how to determine the location and size of the argument vector, and how return values are stored.

A.3. Convex C–1

The Convex C–1 uses a register, the argument pointer, to pass arguments to called subroutines. The calling routine builds an argument vector and sets the argument pointer to point at the base of this vector. Called routines access arguments as offsets from this pointer. Figure A.3 shows the C–1 stack frame at procedure entry, just after the procedure prologue has allocated storage for local variables.

CLASP client routines determine the location of the argument vector from the contents of this register. C–1 compilers store the argument length, as a count of 4 byte words, at the address just below the argument pointer. CLASP client routines determine the argument vector length from the value at this location. CLASP server routines load the argument register with the passed value.

The C–1 stores return values in the *S0* register. This 64 bit register contains all return values.

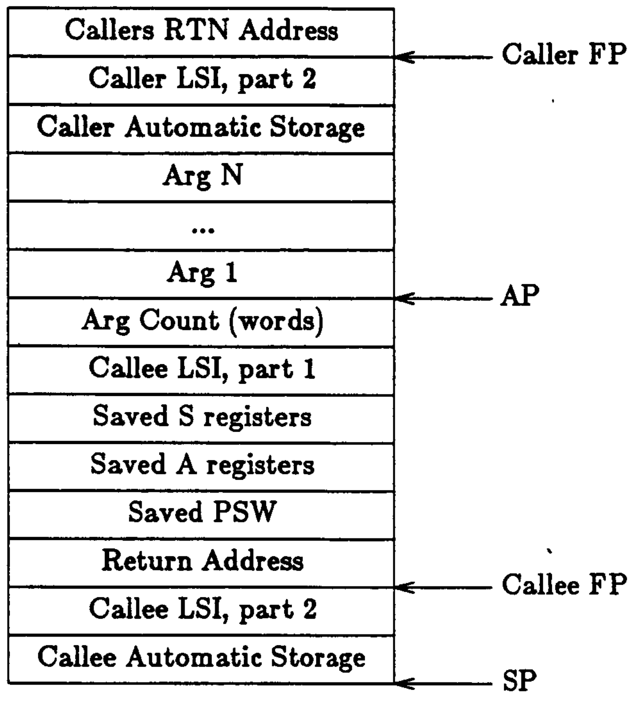

Figure A.3
Convex C–1 Stack Frame
(after procedure prolog)

A.4. IBM RT

The IBM RT–PC presents arguments to subroutines as an array of bytes on the stack. For efficiency reasons, the first 4 arguments are passed in general registers *r2* through *r5* respectively. For subroutines with only a few arguments, this convention improves performance; fewer memory operations are required to pass arguments to the subroutine. For routines that take the address of any of these first four arguments, the called procedure's prologue saves them in a reserved area in the called procedure's stack frame. A multi–word structure might be split; the first several words of the structure may be pased in one or more registers, the rest may be placed on the stack. After the called procedure moves these registers to memory, they form a contiguous argument vector with the other arguments. On the IBM RT, the general register *r1* is used as a stack pointer. The IBM RT stack at procedure entry is shown in figure A.4.

Upon entry, the called procedure adjusts the stack pointer to reserve space for local variables and temporary space. The IBM RT does not use the stack in a true stack–oriented

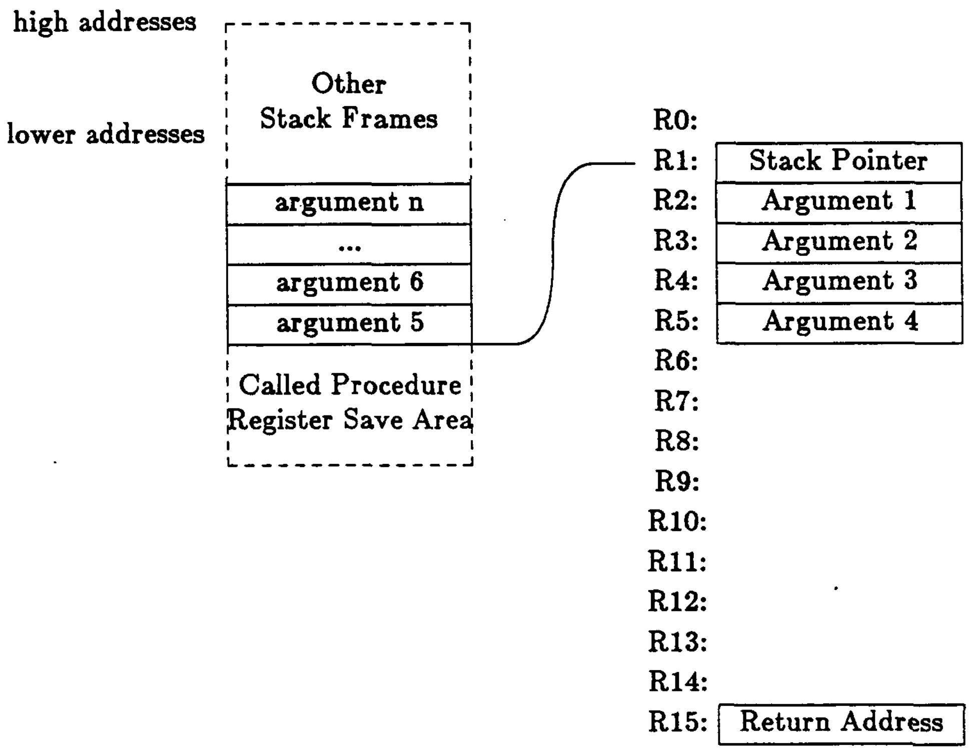

Figure A.4
IBM RT–PC Stack Frame
(at procedure entry)

fashion. Instead of pushing and popping values as needed, the stack is extended to the maximum depth required by the procedure and left there. Local variables are referenced relative to the R13 register; parameters to other subroutines are referenced relative to the R1 register. Figure A.5 shows the stack frame after the procedure prologue has executed.

The IBM RT returns values in registers R2 and R3. Simple values, those of 32 or fewer bits, are completely contained in R2. Floating point values, which are passed as 64 bit quantities, are passed in both R2 and R3.

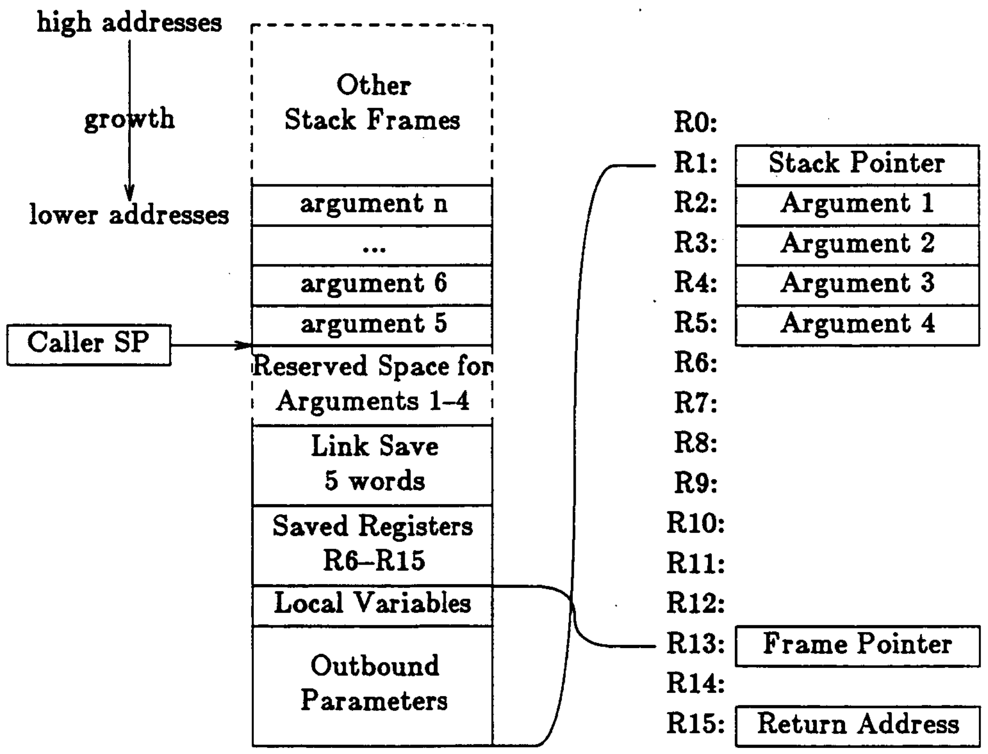

Figure A.5
IBM RT-PC Stack Frame
(after procedure prolog)

Unfortunately for our goals, the compilers on the IBM RT use different conventions for returning structures and passing procedures as formal arguments than we have seen. These differences make a CLASP system between our SUNs and the RT impossible.

APPENDIX B.

PERFORMANCE MEASUREMENTS

This appendix describes the performance of several test programs using our CLASP prototype kernel. We compare the execution times using our CLASP kernel against the execution time for the program with a standard UNIX kernel.

B.1. Double Precision LINPACK Benchmark

We obtained a copy of the LINPACK linear systems library from Argonne National Laboratories [36]. This package contains FORTRAN subroutines to solve the equation:

$$\mathbf{Ax} = \mathbf{b}$$

A benchmark program included with the library performs a number of iterations generating, factoring, and solving a matrix. The **A** and **b** matrices are dimensioned to 200 elements. Another variable determines the size of the system to be solved. We partitioned the FORTRAN program into separate modules. The LINPACK routines to factor and solve the system execute on the server processor. The matrix generation routine and benchmark harness execute on the client processor.

We ran the benchmark for systems whose order ranged from 5 through 200. For each size, we collected the following statistics: client user and system time, server user and system time, total time, page traffic, number of calls, and the time for a non–CLASP version. Each execution made 53 CAPC calls and returns. The number of calls is a function of the benchmark itself, not the size of the system.

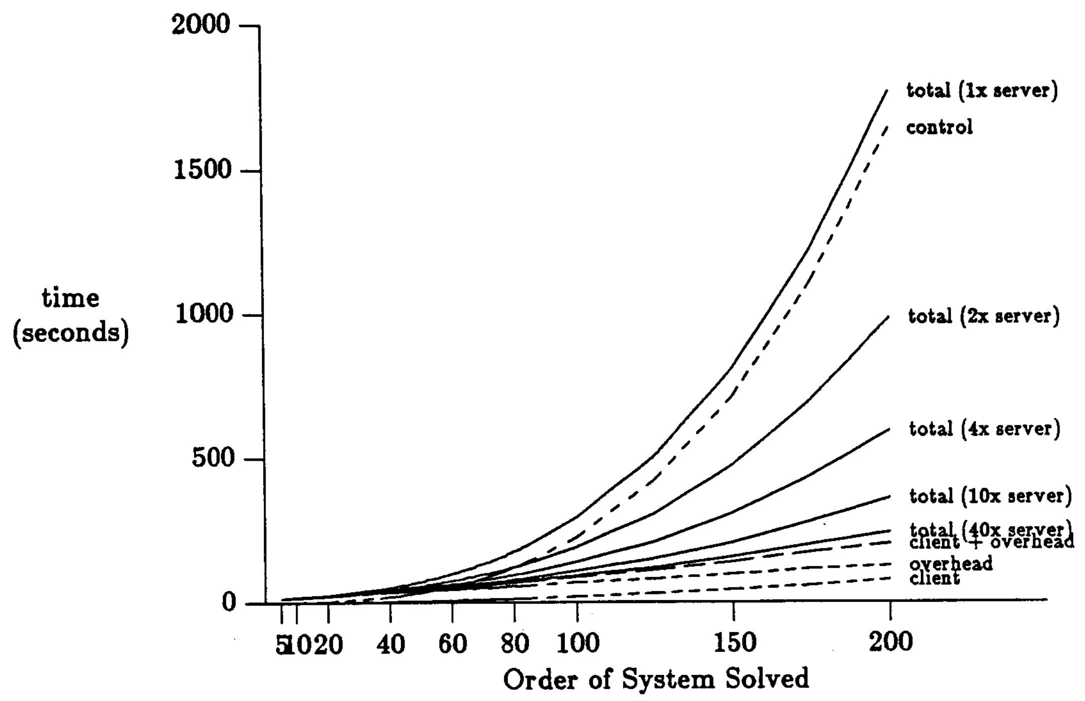

Figure B.1
LINPACK Benchmark Execution Times
Arrays Declared for 200 Elements

The graph in figure B.1 shows the execution times for the LINPACK benchmark. Figure B.3 shows the number of pages moved between client and server for each benchmark run.

Figure B.2 shows the speed ratio between client and server to recover the CLASP overhead. Matrices of smaller order than 59 incur more overhead than can be made up on any server. At 59, the overhead can be offset with a processor that is approximately 15 times faster than the client. When the matrix is of order 81, the server need be only twice as fast as the client to recover the overhead.

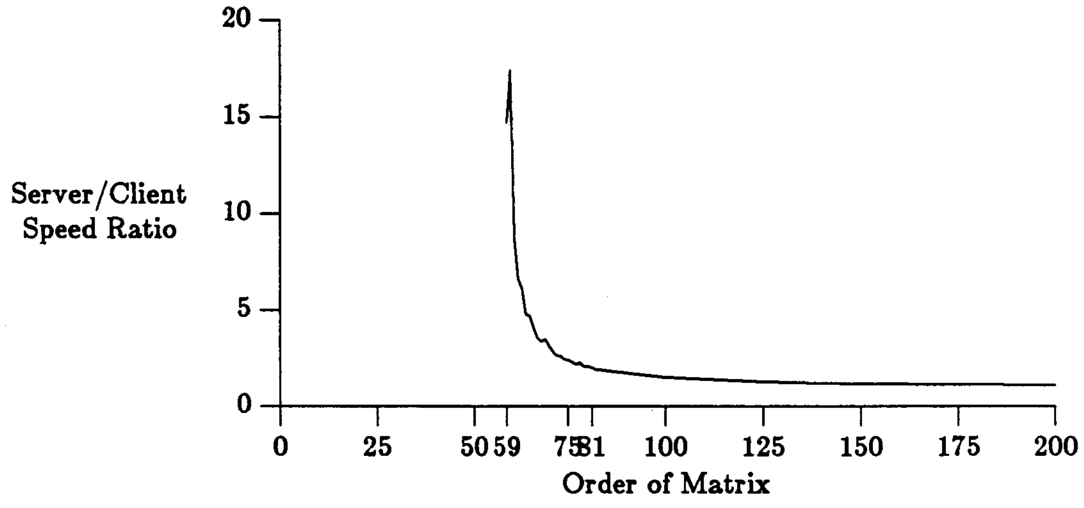

Figure B.2
Speedup required to pay for CAPC overhead
Large dimensioned Arrays

In figure B.3, the paging traffic appears to grow linearly with the size of the system. The overdimensioned matrices interact with the page size to produce this behavior. Each page holds more than one row of the matrix. A page fault moves at least an entire 200 element row, even though only the first 5 columns of that row will be used. Each page holds more than one row of the matrix. For matrices of order 40 through 80, we can discern the steps in the page traffic.

This benchmark uses over–dimensioned arrays. Therefore, array accesses are spread over a larger section of the address space. In the CLASP environment, the sparse use of the address space results in extra paging overhead. Figure B.4 shows the breakeven points for a version of this benchmark that uses arrays dimensioned to the exact size of the problem being solved. Because the smaller array is stored in a more compact section of the address space,

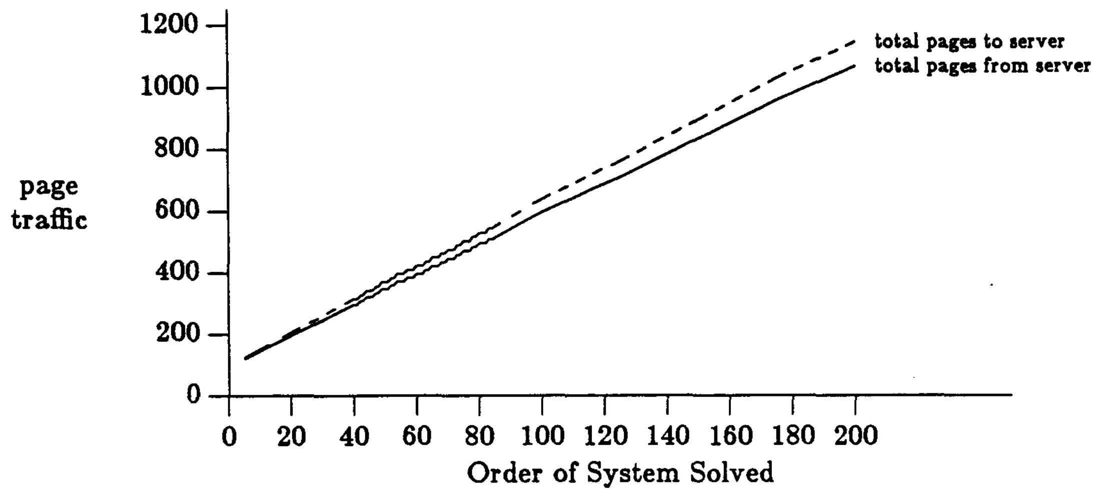

Figure B.3
LINPACK Benchmark Network Paging
Arrays Declared for 200 Elements

the paging traffic is lower than in the original benchmarks. This reduces the breakeven point. With this modification, the CLASP system can break even as soon as the system is order 42 — instead of order 59.

We believe that this benchmark does not show the true advantages of our demand-paging system. The benchmark builds and factors the **A** matrix a number of times. The server does the factoring; the client rebuilds the matrix. This causes additional paging overhead. We feel that a more realistic situation is where the **A** matrix is factored once and then used to solve the system for many different values of **b**.

Figure B.5 shows the pages transferred during virtual time intervals for the execution of the benchmark for a 75x75 matrix. Each timeslot represents 50 milliseconds of processor time. The paging traffic is concentrated in short bursts. Half of the paging traffic occurs in approximately 7% of the virtual time for this program.

Figure B.6 shows the number of times each page moved between processors. This graph does not include the movement of the stack page; the only stack movement was for a single page to the server. Each point on the graph represents a page of 8192 bytes.

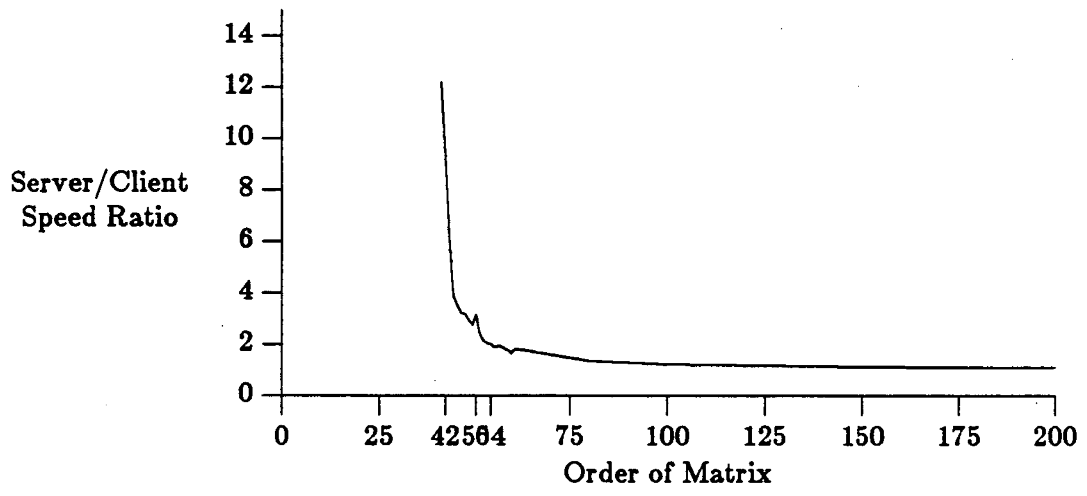

Figure B.4
Server/Client speed ratio to break even
Exact Dimensioned Arrays

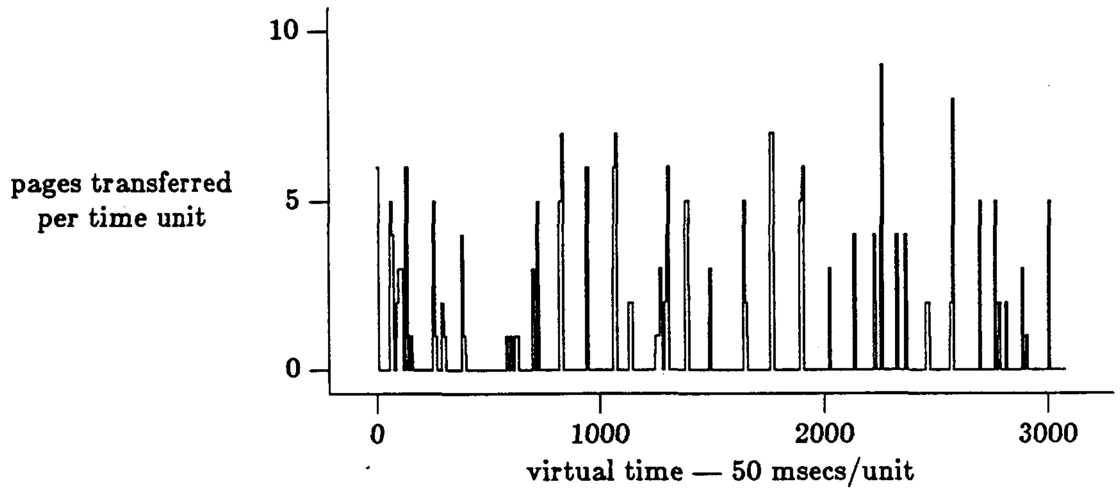

Figure B.5
Linpack Page Transfer Patterns

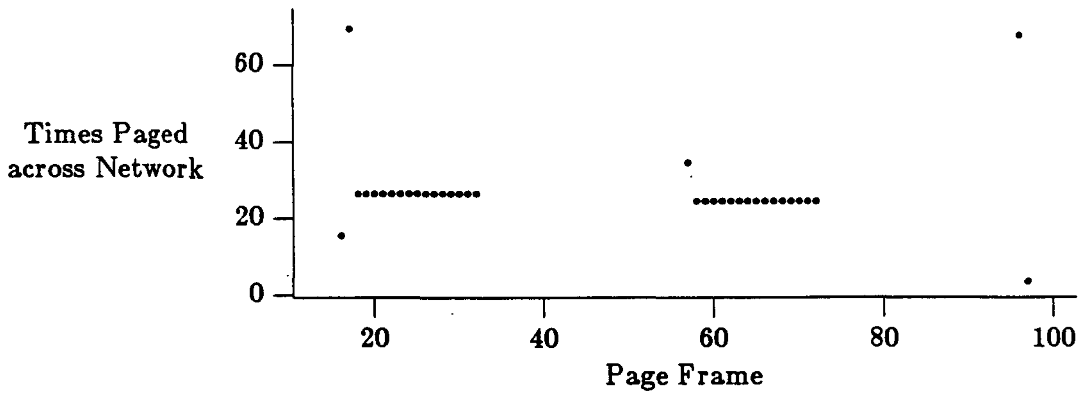

Figure B.6
Page Transfer Frequency

B.2. Compression Program

We partitioned the *compress* program from the 4.3 BSD distribution into client and server routines [87]. This program uses a modified Lempel–Ziv algorithm to generate codes for common substrings and replace them in the compressed file.

A profiling run showed that the program spends much of its time in two routines: *compress*() and *output*(). We built a version of the program with these two routines on the server architecture.

We used the partitioned program to compress a copy of our UNIX kernel. The program compressed this 472,689 byte file into 296,314 bytes. We used the UNIX *gprof*(1) utility to determine what portions of the program used the most CPU time. The *compress*() function was invoked once and used 11.7 seconds of CPU time. The *output*() function was invoked 159,280 times and used another 7.13 seconds of CPU time. Although each invocation of the *output*() subroutine was too short to make a CAPC advantageous, all but two of these invocations came from the *compress*() routine — which is on the same processor as the output routine. *Output*() made a small number of calls to routines on the client processor (39 calls in this instance).

We built a version of the compress program with the *compress*() and *output*() functions on the server processor. For the data files we ran, the partitioned program's overhead was larger than the execution time of the original program.

Figure B.7 shows which pages were moved and how often. Only two stack pages moved — each moved 1 time. The stack frames are not included in this graph. Page 16 moves because most of the static variables are on that page. The client and server routines are accessing different variables that happen to be on the same page. A more sophisticated loader might place these variables on different pages to reduce this contention.

Figure B.8 shows the actual paging behavior of the program. This depicts the pages transferred during each time slot. Each timeslot is 50 milliseconds of user time; the time spent transferring pages between hosts is not included. Like the partitioned LINPACK benchmark, this program generated most of its network page faults in a short time period. Half of the page faults were generated in approximately 5% of the virtual time.

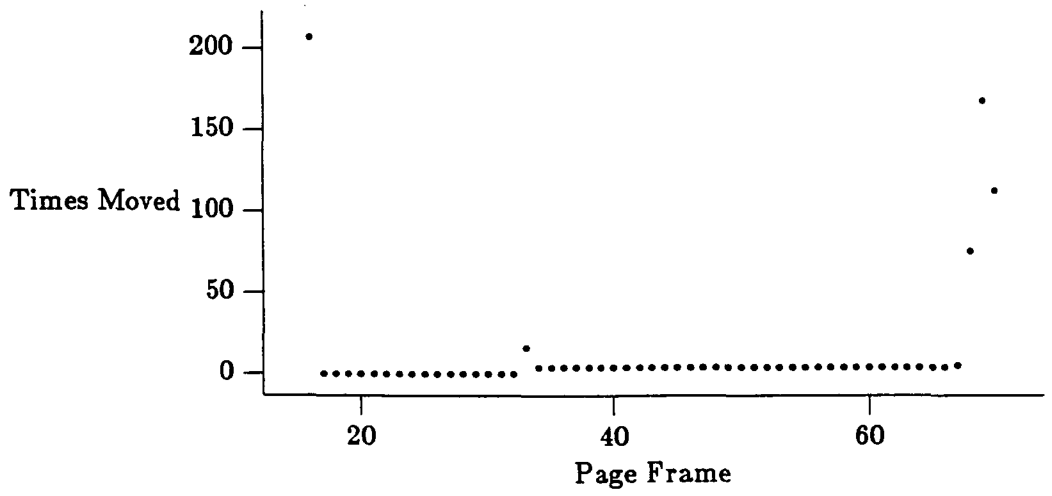

Figure B.7
Page Transfer Frequency

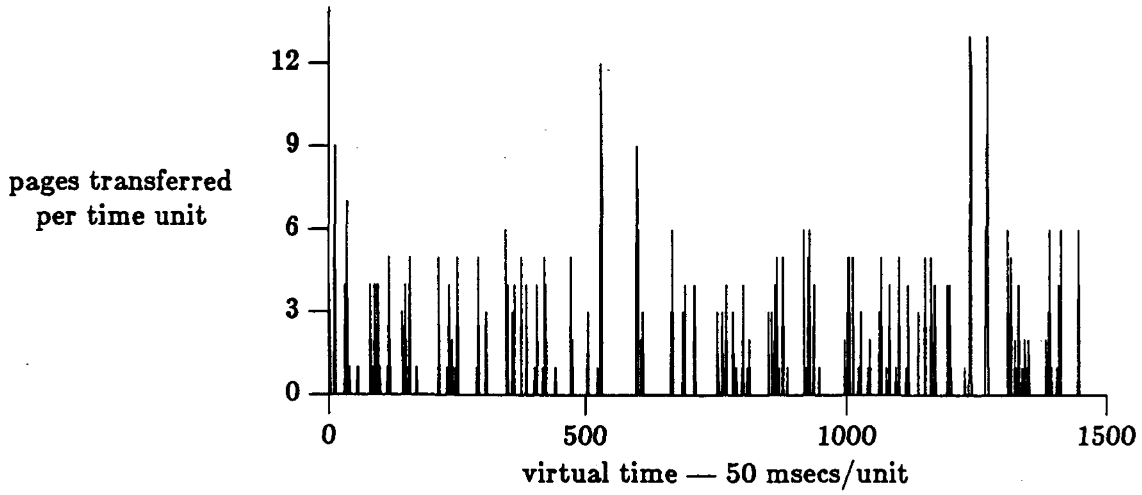

Figure B.8
Page Transfer Patterns

APPENDIX C.

CODE SAMPLES

This appendix contains sample code segments for routines that demonstrate some features of the CLASP architecture.

The first sample shows how local and remote procedures can be passed as formal paramters to other procedures. There are no special actions to differentiate between local and remote procedures.

The second example shows a program that builds and traverses a tree structure. The program passes pointers between the client and server; as the program traverses the tree, the demand paging system moves parts of the tree as the program accesses them. The routines have the same structure and arguments as they would if compiled for a more traditional single–processor system.

C.1. Procedures as Formal Parameters

This example demonstrates how the CLASP system allows the applications to pass procedures as formal parameters. No special compilation techniques are required to account for client and server differences. Both local and remote procedures are stored in the argument list using the same representation. The called procedure does not require special operations to differentiate between local and remote formal procedures. Figures C.1 and C.2 show the client and server portions of a program that passes procedures as formal parameters.

```
extern int  foosquare (), foocube (), foo ();

main ()
{
    int     i, j;

    for (i = 0; i < 10; i++)
    {
        j = foo (foosquare, i);
        j = foo (foocube, i);
    }
    exit (0);
}

int     foosquare (i) int   i;
{
    return (i * i);
}
```

Figure C.1
Formal Procedures — Client Side

```
int     foo (proc, arg)
int     (*proc) ();              /* procedure parm */
int     arg;
{
    return ((*proc) (arg));
}

foocube (arg) int     arg;
{
    return (arg * arg * arg);
}
```

Figure C.2
Formal Procedures — Server Side

C.2. Pointer Structures

This sample program builds and traverses binary trees. Some of the tree manipulation routines execute on the client; others execute on the server processor. The code in this example was written for a single processor system.

There were no changes to the source code to make it run in a CAPC environment. We only changed the linking phase of the compilation process to use our new loader. Figure C.3 contains the main section of the program, which executes on the client. The code in figure C.4 performs several operations on the tree. This code also executes on the client. The code in figure C.5 traverses the tree in postorder and preorder. These two routines execute on the server processor.

```
main (argc, argv)
int     argc;
char  **argv;
{
    int     i, value, parms;
    char    buf[128], cmd;
    static struct node  root;

    while (printf ("CMD: "), fflush (stdout), (gets (buf) != NULL))
    {
        if (strlen (buf) == 0) break;
        parms = sscanf (buf, "%c %d", &cmd, &value);
        switch (cmd)
        {
            case 'I':
                inorder (&root); break;
            case 'L':
                preorder (&root); break;
            case 'R':
                postorder (&root); break;
            case 'A':
                i = insert (value, &root);
                printf ("value %d, now has %d hits\n", value, i);
                break;
            case 'F':
                i = find (value, &root);
                printf ("value %d has %d hits\n", value, i);
                break;
            case 'Q':
                goto quit;
        }
    }
quit:
    exit (0);
}
```

Figure C.3
Pointers in a CAPC environment — Main code

```
find (value, root) int value; struct node *root;
{
    if (root == (struct node *) NULL) return (-1);
    if (root -> value == value) return root -> hits;
    if (root -> value > value) return find (value, root -> left);
    if (root -> value < value) return find (value, root -> right);
    return (-1);
}

insert (value, root) int value; struct node *root;
{
    if (root == (struct node *) NULL) exit (1);
    if (value == root -> value) return (++(root -> hits));
    if (value < root -> value) {                /* down left side */
        if (root -> left == (struct node *) NULL) {
            root -> left = (struct node *)malloc (sizeof (struct node));
            root -> left -> value = value;
            return (root -> left -> hits = 1);
        }
        return insert (value, root -> left);    /* recurse */
    }
    if (value > root -> value) {                /* down right side */
        if (root -> right == (struct node *) NULL) {
            root -> right = (struct node *)malloc (sizeof (struct node));
            root -> right -> value = value;
            return (root -> right -> hits = 1);
        }
        return insert (value, root -> right);   /* recurse */
    }
    return (-1);
}

inorder (root) struct node *root;
{
    if (root == (struct node *) NULL) return;
    inorder (root -> left);
    printf ("%d: %d hits\n", root -> value, root -> hits);
    inorder (root -> right);
}
```

Figure C.4
Pointers in a CAPC environment — Client Code

```
#include        <stdio.h>
#include        "node.h"

postorder (root)
struct node *root;
{
    if (root == (struct node *) NULL)
        return;
    postorder (root -> left);
    postorder (root -> right);
    printf ("%d: %d hits\n", root -> value, root -> hits);
}

preorder (root)
struct node *root;
{
    if (root == (struct node *) NULL)
        return;
    printf ("%d: %d hits\n", root -> value, root -> hits);
    preorder (root -> left);
    preorder (root -> right);
}
```

Figure C.5
Pointers in a CAPC environment — Server Code

APPENDIX D.

CLASP CONFIGURATION AND LOG FILES

Our prototype uses a static configuration table to assign server processors. The file */usr/local/etc/claspd.config* contains the configuration data. The file contains lines that describe how much logging information to generate and addresses for servers of appropriate architectures. Figure D.1 shows a sample configuration file.

Our configurations usually store logging information in the files */usr/adm/claspd.log* and */usr/adm/claspd.prof.* The *claspd.log* file contains high level information describing the current host addresses for specific architectures and the starting and finishing times for server processes. Figure D.2 shows a segment from this file.

Claspd.prof provides more detailed information. This file records page traffic and call behavior. At normal logging levels, the kernel stores summary data in this file. For each client and server on the local host, the file contains the number of CAPC calls and returns and the number of pages moved across the network. More detailed logging generates a line for each CAPC call, CAPC return or page transfer. All lines are marked with the current time and process identifier. A segment of this file is shown in figure D.3

```
# configuration file for CLASP kernel.
# The daemon reads this file at startup and whenver it receives
# a SIGHUP signal.
#
# Syntax:
#
#       profiling       level           hostname
#               missing hostname defaults to localhost
#               missing level defaults to 2
#
#       logging {on|off}        pathname
#               missing pathname leaves it unchanged.
#               must specify on/off field.
#
profiling       2       brutus.cs.uiuc.edu
logging         on      /usr/adm/claspd.log

#
#       server  architecture    hostname
#               architecture is integer
#               hostname is string
#       gotta specify both.
#
# 10 == M_RBE1
server  10      crl.cs.uiuc.edu

# 20 == M_68020R
server  20      brutus.cs.uiuc.edu
```

Figure D.1
Sample Claspd Configuration File
/usr/local/etc/claspd.config

```
Fri Mar 27 11:04:15 1987: daemon     88: Re-initialize server tables
Fri Mar 27 11:04:17 1987: daemon     88: 0x10 @ brutus.cs.uiuc.edu
Fri Mar 27 11:04:19 1987: daemon     88: 0x20 @ brutus.cs.uiuc.edu
Fri Mar 27 20:55:05 1987: server   1274: client at 192.17.238.2/1053
Fri Mar 27 20:55:19 1987: server   1274: exit/sig 0/11.
                 user/sys 0.100000/6.960000 secs
Sun Mar 29 11:29:39 1987: server   3148: client at 192.17.238.2/1137
Sun Mar 29 11:30:11 1987: server   3148: exit/sig 0/11.
                 user/sys 0.000000/0.960000 secs
```

Figure D.2
Sample claspd.log

```
037398/400001 pid 3147: call to server at     0x73f8
037398/400003 pid 3147: pageout at 0xeffc000
037398/460002 pid 3147: pageout at 0x20000
037398/520001 pid 3147: pageout at 0x28000
037398/580000 pid 3147: pageout at 0x2a000
037398/640000 pid 3147: call from server to   0x326c
037398/640001 pid 3147: pagein  at 0x20000
037398/700000 pid 3147: pagein  at 0x28000
037398/740001 pid 3147: return to server at   0x7448
037398/760002 pid 3147: pageout at 0x28000
037398/820000 pid 3147: pageout at 0x2c000
037398/860001 pid 3147: call from server to   0x326c
037398/860002 pid 3147: pagein  at 0x28000
037398/920001 pid 3147: return to server at   0x7448
037398/940000 pid 3147: pageout at 0x28000
037398/980001 pid 3147: call from server to   0x326c
037399/000000 pid 3147: pagein  at 0x28000
037399/040001 pid 3147: return to server at   0x7448
037399/060001 pid 3147: pageout at 0x20000
...
037409/420000 pid 3147: return from server to 0x21f8
037411/020000 pid 3147: r1: CAPC calls/returns: local 0/0  network 2/9
037411/020001 pid 3147: r1: CAPC pageins 11, pageouts 15
```

Figure D.3
Sample claspd.log

REFERENCES

1. "IEEE Floating Point Standard #754", IEEE.

2. *XNS Courier under UNIX.* In: **UNIX Programmer's Manual, 4.3 Berkeley Software Distribution, Virtual VAX–11 Version.**

3. "Courier: The Remote Procedure Call Protocol", Xerox System Integration Standard 038112 Xerox Corporation, Stamford, Connecticut, 1981.

4. **MC68000 16–bit Microprocessor User's Manual.** Prentice–Hall, Englewood Cliffs, NJ, 1982.

5. "Pyramid Processor Architecture Manual (Preliminary)", Pyramid Technology Corporation, 1983.

6. **MC68020 32–bit Microprocessor User's Manual.** Prentice–Hall, Englewood Cliffs, NJ, 1984.

7. "CONVEX Architecture Handbook", Convex Computer Corporation, 1985.

8. "DDN Protocol Handbook", NIC 50004, SRI International, 1985.

9. "DOMAIN Architecture: A Technical Overview", Apollo Computer Inc., 1985.

10. "DOMAIN Series 3000 Technical Reference Hardware Architecture Handbook", Apollo Computer Inc., 1985.

11. "FX/Series Architecture Manual", Alliant Computer Systems Corporation, 1985.

12. "Balance Technical Summary", Sequent Computer Systems, Inc., 1986.

13. "External Data Representation Protocol Specification", Sun Microsystems, Inc., 1986, p. 61.

14. "Multimax Technical Summary", Encore Computer Corporation, 1986.

15. "Network File System Protocol Specification", Sun Microsystems, Inc., 1986, p. 32.

16. "Remote Procedure Call Programming Guide", Sun Microsystems, Inc., 1986, p. 69.

17. "Remote Procedure Call Protocol Specification", Sun Microsystems, Inc., 1986, p. 26.

18. *The Remote Virtual Disk System.* In: **Academic Information Systems 4.2 for the IBM RT PC.**, 1986, pp. 185–234.

19. "RPCL, A Remote Procedure Call Language", Sun Microsystems, Inc., 1986.

20. "System Administration for the Sun Workstation", Sun Microsystems, Inc., 1986.

21. "Writing Device Drivers for the Sun Workstation", Sun Microsystems, Inc., 1986.

22. Accetta, Mike, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Michael Young. *Mach: A New Kernel Foundation for UNIX development.* **USENIX Conference Proceedings** (June 1986) pp. 93–111.

23. Almes, Guy T., Andrew P. Black, Edward D. Lazowska and Jerre D. Noe. "The Eden System: A Technical Review", Technical Report 83–10–05, Department of Computer Science, University of Washington, Seattle, Washington 98195, 1983, p. 25.

24. Batson, A. P. *Program behavior at the symbolic level.* **Computer** (November 1976) vol. 9, no. 11, pp. 21–28.

25. Batson, A. P. and W. Madison. *Measurements of major locality phases in symbolic reference strings.* **Proc. Int. Symp. Comput. Performance Modeling, Measurement, and Evaluation** (March 1976) pp. 75–84.

26. Brown, Mark R., Karen N. Kolling and Edward A. Taft. *The Alpine File System.* **ACM Transactions on Computer Systems** (November 1985) vol. 3, no. 4, pp. 261–293.

27. Brownbridge, D. R., L. F. Marshall and B. Randell. *The Newcastle Connection, or UNIXes of the World Unite!.* **Software – Practice and Experience** (1982) pp. 1147–1162.

28. Campbell, Roy H., Gary M. Johnston and Vince F. Russo. "CHOICES: a Class Hierarchical Open Interface for Custom Embedded Systems", unpublished, 1987.

29. Cheriton, David R. and Michael Stumm. "The Multi–Satellite Star: Structuring Parallel Computations for a Workstation Cluster", Stanford University, p. 31.

30. Cheriton, David R. and Willy Zwaenepoel. "The Distributed V Kernel and its Performance for Diskless Workstations", Stanford University, 1983.

31. Denning, Peter J. *The Working Set Model for Program Behavior.* **Communications of the ACM** (May 1968) vol. 11, no. 5, pp. 323–333.

32. ——. *Working Sets Past and Present.* **IEEE Transactions on Software Engineering** (January 1980) vol. SE–6, no. 1, pp. 64–84.

33. Denning, Peter J. and Kevin C. Kahn. *A study of program locality and lifetime functions.* **Proceedings of the 5th ACM Symposium on Operating Systems Principles** (November 1975) pp. 207–216.

34. Devarakonda, M. V., R. E. McGrath, R. H. Campbell and W. J. Kubitz. *Networking a Large Number of Workstations Using Unix United.* **Proc. IEEE Computer Workstations Conference** (November 1985).

35. Ditzel, David R. and H. R. McLellan. *Register Allocation for Free: The C Machine Stack Cache.* **SIGARCH** (March 1982) vol. 10, no. 2, pp. 48–56.

36. Dongarra, J., C. Moler, J. Bunch and G. Stewart. "LINPACK Users Guide", SIAM, 1979.

37. Donnelly, Jeffrey M. "Porting the Newcastle Connection to 4.2 BSD", University of Illinois, Urbana, 1985, p. 37.

38. Ezzat, A. K. "Decentralized Control of Distributed Processing Systems", Ph.D. dissertation, University of New Hampshire, Durham, NH, 1982.

39. Ezzat, Ahmed K., R. Daniel Bergeron and John L. Pokoski. *Task Allocation Heuristics for Distributed Computing Systems.* **The 6th International Conference on Distributed Computing Systems** (May 1986) pp. 337–346.

40. Feldman, S. I. "Make — A Program for Maintaining Computer Programs", Bell Laboratories, 1978.

41. Fitzgerald, Robert and Richard F. Rashid. *The Integration of Virtual Memory Management and Interprocess Communcation in Accent.* **ACM Transactions on Computer Systems** (May 1986) vol. 4, no. 2, pp. 147–177.

42. Gajski, Daniel, David Kuck, Duncan Lawrie and Ahmed Sameh. "Construction of a Large Scale Multiprocessor", Department of Computer Science Technical Report #1123, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1983, p. 36.

43. Gancarz, Michael. *Uwm: A User Interface for X Windows.* **Proceedings of the Summer '86 Usenix Conference** pp. 429–440.

44. Gettys, James. *Problems Implementing Window Systems in Unix.* **1986 Winter USENIX Technical Conference** (January 15, 1986) pp. 89–97.

45. Gettys, Jim, Ron Newman and Tony Della Fera. *Xlib – C Language X Interface: Protocol Version 10.* (November 16, 1986).

46. Hansen, Per Brinch. *Distributed Processes: A Concurrent Programming Concept.* **Communications of the ACM** (November 1978) vol. 21, no. 11, pp. 934–941.

47. Henry, G. Glenn. *IBM RT PC Architecture and Design Decisions.* In: **RT Personal Compuer Technology**, Frank Waters, ed., 1986, pp. 2–5.

48. Hester, P. D., Richard O. Simpson and Albert Chang. *The IBM RT PC ROMP and Memory Management Unit Architecture.* In: **RT Personal Computer Technology**, Frank Waters, ed., 1986, pp. 48–56.

49. Hopkins, M. E. *Compiling for the RT PC ROMP.* In: **RT Personal Computer Technology**, Frank Waters, ed., 1986, pp. 76–82.

50. Horton, Kurt H. "Multicomputer Interconnection Using Work Parallel Shift Register Ring Networks", PhD. Thesis, Department of Computer Science Technical Report #1164, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1984, p. 79.

51. Jamp, R. and J. R. Spirn. "VMIN based determination of program macro–behavior", Department of Computer Science, Pennsylvania State University, University Park, 1979.

52. Johnson, S. C. *A Tour Through the Portable C Compiler.* In: **UNIX Programmer's Manual, Seventh Edition, Vol. 2A.** Bell Telephone Laboratories Incorporated, Murray Hill, New Jersey, p. 15.

53. Kahn, K. C. "Program behavior and load dependent system performance", Ph.D. dissertation, Dept. of Computer Science, Purdue University, West Lafayette, IN, 1976.

54. Kevorkian, D. E. (ed.). **System V Interface Definition.** AT&T, 1985.

55. Kolstad, Robert Bruce. "Distributed Path Pascal: A Language for Programming Coupled Systems", PhD. Thesis, Department of Computer Science Technical Report #1136, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1983, p. 75.

56. Kolstad, Rob. private communications, 1986.

57. Kronenberg, Nancy P., Henry M. Levy and William D. Strecker. *VAXclusters: A Closely-Coupled Distributed System.* **ACM Transactions on Computer Systems** (May 1986) vol. 4, no. 2, pp. 130–146.

58. Kuck, Sharon M., David A. McNabb, Stephen V. Rice and Yehoshua Sagiv. "The PARAFRASE Database User's Manual", Department of Computer Science Technical Report #1046, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980, p. 30.

59. Lauer, Hugh C. and Roger M. Needham. *On the Duality of Operating System Structures.* In: **Operating Systems: Theory and Practice**, D. Lanciaux, ed. North Holland Publishing Company, 1979, pp. 371–384.

60. Li, Kai. "Shared Virtual Memory on Lossely-coupled Multiprocessors", Ph.D. Thesis, Yale University, 1986.

61. Li, Kai and Paul Hudak. *Memory Coherence in Shared Virtual Memory Systems.* **Proceedings of the Fifth Annaul ACM Syposium on Principles of Distributed Computing** (August 1986) pp. 229–239.

62. Lions, J. "Notes on the UNIX Operating System", University of New South Wales, 1976.

63. Madison, A. W. and A. P. Batson. *Characteristics of program localities.* **Communications of the ACM** (May 1976) vol. 19, pp. 285–294.

64. Metcalfe, Robert M. and David R. Boggs. *Ethernet: Distributed Packet Switching for Local Computer Networks.* **Communications of the ACM** (July 1976) vol. 19, no. 7.

65. Mitchell, James G. and Jeremy Dion. *A Comparison of Two Network-Based File Servers.* **Communications of the ACM** (April 1982) vol. 25, no. 4, pp. 233–245.

66. Nelson, Bruce Jay. "Remote Procedure Call", Carnegie-Mellon University, 1981, p. 201.

67. O'Quin, J. C. *The IBM RT PC Subroutine Linkage Convention.* In: **RT Personal**

Computer Technology, Frank Waters, ed., 1986, pp. 131–133.

68. Ousterhout, John K., Donald A. Scelza and Pradeep S. Sindhu. *Medusa: An Experiment in Distributed Operating System Structure.* **Proceedings of the Seventh ACM Symposium on Operating Systems Principles** (December 1979) pp. 115–116.

69. ——. *Medusa: An Experiment in Distributed Operating System Structure.* **Communications of the ACM** (February 1980) vol. 23, no. 2, pp. 92–105.

70. Patterson, David A. *Reduced Instruction Set Computers.* **Communications of the ACM** (January 1985) vol. 28, no. 1, pp. 8–22.

71. Peterson, James L. *Petri Nets.* **ACM Computing Surveys** (September 1977) vol. 9, no. 3, pp. 223–252.

72. Popek, Gerald J., Bruce J. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin and G. Thiel. *LOCUS: A Network Transparent, High Reliability Distributed System.* **Proceedings of the Eighth ACM Symposium on Operating Systems Principles, Pacific Grove, California** (December 1981) pp. 169–177.

73. Postel, Jon B. "Transmission Control Protocol - DARPA Internet Program Protocol Specification", RFC 793, USC/Information Sciences Institute, 1981, p. 85.

74. Rashid, Richard F. *Threads of a New System.* **UNIX Review** (August 1986) vol. 4, no. 8, pp. 37–49.

75. Rashid, Richard F. and George G. Robertson. *Accent: A communication oriented network operating system kernel.* **ACM SIGOPS** (December 1981) vol. 15, no. 5, pp. 64–75.

76. Rogers, Gregory Scott. "UIGKS: A Distributed Graphics System", Department of Computer Science Technical Report #1253, University of Illinois at Urbana–Champaign, Urbana, Illinois, 1986, p. 47.

77. Russo, Vincent Frank. "LINK: A Kernel Based Distributed UNIX", University of Illinois, Urbana, 1987.

78. Spector, Alfred Z. "Multiprocessing Architectures for Local Computer Networks", Stanford University, 1981, p. 116.

79. Stone, Harold S. and Shahid H. Bokhari. *Control of Distributed Processes.* **IEEE Computer** (July 1978) pp. 97–106.

80. Swinehart, Daniel C., Polle T. Zellweger, Richard J. Beach and Robert B. Hagmann. *A Structural View of the Cedar Programming Environment.* **ACM Transactions on Programming Languages and Systems** (October 1986) vol. 8, no. 4, pp. 419–490.

81. Tanenbaum, Andrew S. and Sape J. Mullender. *An Overview of the Amoeba Distributed Operating System.* **ACM Operating Systems Review** (July 1981) vol. 15, no. 3, pp. 51–64.

82. Taylor, Bradley and David Goldberg. *Secure Networking in the Sun Environment.* **Proceedings of the Summer 1986 USENIX Technical Conference** (June 1986) pp. 28–37.

83. Theimer, Marvin M., Keith A. Lantz and David R. Cheriton. *Preemptable Remote Execution Facilities for the V-System.* **ACM Operating Systems Review** (December 1985) vol. 19, no. 5, pp. 2–12.

84. Walker, Bruce J., Gerald J. Popek and R. English. *The LOCUS Distributed Operating System.* **SIGOPS** (October 1983) vol. 17–5, pp. 49–70.

85. Walsh, Dan, Bob Lyon, Gary Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg and P. Weiss. *Overview of the Sun Network File System.* **USENIX Conference Proceedings** (January 1985) pp. 117–124.

86. Watson, Tom. private communications, 1987.

87. Welch, Terry A. *A Technique for High Performance Data Compression.* **IEEE Computer** (June 1984) vol. 17, no. 6, pp. 8–19.

88. Wulf, William A. and C. G. Bell. *C.mmp — A multi-mini-processor.* **Proc. AFIPS 1972, FJCC** (December 1972) vol. 41, pp. 765–777.

89. Wulf, William A., E. Cohen, William M. Corwin, Anita K. Jones, R. Levin, C. Pierson and F. J. Pollack. *HYDRA: The Kernal of a Multiprocessor Operating System.* **Communications of the ACM** (June 1974) vol. 17, no. 6, pp. 337–345.

90. Wulf, William A., Roy Levin and Samuel P. Harbison,. **HYDRA/C.mmp.**, 1981.

91. Wulf, William A., R. Levin and C. Pierson. *Overview of the Hydra Operating System Development.* **Proceedings of the Fifth Symposium on Operating Systems Principles, University of Texas at Austin** (November 1975) pp. 122–131.

VITA

Raymond Brooke Essick IV was born in [redacted] on [redacted] [redacted]. He entered the University of Illinois at Urbana–Champaign in August 1977 and received a B.S. in computer science in January 1981. During his undergraduate career, Ray was a member of the University's swimming team and earned varsity letters for each of his four years on the team.

Ray began his graduate studies at the University of Illinois at Urbana–Champaign in January 1981. In May 1983, Ray received the M.S. degree in Computer Science. He received his Ph.D. in Computer Science from the University of Illinois in May 1987. During his years in graduate school, Ray took an active part in the support of the Computer Science Department's UNIX systems.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-87-1340	2.	3. Recipient's Accession No.

4. Title and Subtitle THE CROSS-ARCHITECTURE PROCEDURE CALL	5. Report Date May 21, 1987
	6.
7. Author(s) Raymond Brooke Essick IV	8. Performing Organization Rept. No. R-87-1340
9. Performing Organization Name and Address Department of Computer Science 1304 W. Springfield Urbana, IL 61801	10. Project/Task/Work Unit No.
	11. Contract/Grant No. NASA NSG 1471
12. Sponsoring Organization Name and Address NASA Langley Research Center Hampton, VA 23665	13. Type of Report & Period Covered Ph.D. Thesis
	14.

15. Supplementary Notes

16. Abstracts This thesis introduces the Cross-Architecture Procedure Call. Cross-Architecture Procedure Calls (or CAPCs) combine virtual memory, high speed networking, and compatible data representations to accelerate an application's computations without modifying its code. CAPCs allow workstations to use, on a demand basis, faster or more expensive processors as compute servers so that each of an applications functions can be executed by the most appropriate processor.
The CAPC process executes in a single virtual address space shared by several CPUs. Instructions for each CPU are stored in different regions of the virtual address space. Routines are compiled for the processor that can most effectively execute them. CAPCs do not require special calling sequences to transfer control between processors. Instead, virtual memory page protections are used to implement transfers between processors. Routines on all processors share access to global variables, including pointer data types. A modified operating system uses demand-paging to control access to these shared pages. Because the CPUs share the same internal data representation, pages can be moved between processors without any conversion operations.
This thesis describes the CAPC construct, a sufficient level of similarity between processors architectures to use CAPCs, and a CAPC implementation based on the SUN 3.0 Operating System.

17. Key Words and Document Analysis. 17a. Descriptors

transparent compute servers
remote procedure call
loosely coupled virtual memory
distributed processing
distributed sequential process

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 152
	20. Security Class (This Page UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70) USCOMM-DC 40329-P71