The Walnut Kernel:
A Password-Capability Based Operating System

Thesis submitted for examination
for the Degree of
Doctor of Philosophy,
Department of Computer Science
Monash University
January 1996

Maurice David Castro
B.Sc.(Hons), Monash University

Abstract

This thesis describes both a capability based kernel - the Walnut Kernel - and hardware designed to support that kernel. The kernel provides an environment in which programs written and operated by mutually antagonistic users can co-exist securely. A Password-Capability mechanism is used to provide access control to objects. The hardware is designed to support cost efficient expansion of the number of processors within a multiprocessor.

The Walnut Kernel was designed to be portable to a wide range of microprocessors and memory architectures. The kernel avoids using processor specific features where there are more widely available mechanisms. Paged memory management was adopted as the basic access control mechanism because of the large number of processors which support it. This resulted in a page sized protection granularity. The architecture of the kernel was designed to scale from uniprocessors to large multiprocessors. To accommodate this design requirement, device drivers on uniprocessor systems use a shared memory page to communicate with the kernel. This is equivalent to special purpose processors sharing memory with a master processor on a multiprocessor. The efficiency of multiprocessor implementations was increased by decreasing interprocessor communication. Page tables and other system data are periodically expired and new tables constructed. This practice of timing out data ensures that local data is kept up-to-date, and only a minimal amount of local state is retained. Furthermore this practice eliminates the need to inform other processors of changes in local information.

Two implementations of the Walnut Kernel are currently in service. One version operates in an emulated environment under a host operating system. The other version operates on i486 based IBM PCs. The performance of the latter version compares well with contemporary operating systems.

The kernel differs from earlier password-capability based systems in that it introduces operators for the selective removal of rights from a capability, and mechanisms which restrict the use of a capability to a specific process. Message-passing and subprocess mechanisms have been introduced to enhance the handling of asynchronous events. The changes were motivated by the requirements of programmers using the

system.

Application programs have been written to demonstrate the features of the kernel. Included among these user level programs are managers for floppy diskette drives, and, for the screen and keyboard. The programs and the programming techniques used are described.

The proposed hardware has eliminated centralised switching devices in favor of distributing the processor interconnection hardware across the nodes of the multiprocessor. Each processor node has its own clock which removes the physical constraints associated with a centralised clock.

The kernel and the proposed hardware are both part of the Secure RISC Architecture project of the Department of Computer Science, Monash University.

Acknowledgments

I would like to thank Professor Chris Wallace for supervising the work described in this thesis. His assistance, suggestions and guidance throughout the project is gratefully acknowledged.

Many of the staff and postgraduate students of the Department of Computer Science have contributed to work related to the Walnut Kernel and the proposed hardware. In particular, I would like to acknowledge: Mr Glen Pringle for programming work conducted on both the Walnut Kernel and user level programs; Mr Carlo Kopp for his work on user level libraries; Dr Ronald Pose for work relating to the proposed hardware and a constant stream of suggestions for kernel features; and Mr Gerhard Fries and Mr David Duke for their technical assistance in areas relating to hardware.

The financial assistance provided by an Australian Postgraduate Research Award has been appreciated.

The 'Secure RISC Architecture' project was supported by a grant from the Australian Research Council (A49030623).

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university or other institution.

To the best of my knowledge, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Where the work in this thesis is based upon joint research, this thesis discloses the relative contributions of the respective authors.

Maurice D Castro

Department of Computer Science,
Monash University,
Clayton, Victoria, 3168.
Australia.

January 31, 1997

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes a capability based operating system developed in the Department of Computer Science at Monash University. The operating system is commonly known as the Walnut Kernel.

In addition to being an independent project to develop a portable capability based operating system, the Walnut Kernel forms the software basis of a major project within the Department to develop a scalable multiprocessor system. The Secure RISC Architecture project builds on work done towards the development of the Monash Multiprocessor (also known as the Password-Capability System) [APW86, And87, Pos91, APW85, AW85] which resulted in a shared memory multiprocessor system with a novel capability based operating system. The Secure RISC Architecture and the Walnut Kernel were inspired by the concepts behind the Monash Multiprocessor; however, due to limitations imposed on the original design, the two new projects started with the successful concepts imparted from their predecessors.

1.1 Overview

The Password-Capability System - an ancestor of the current project - is discussed in chapter 2. The kernel and the purpose built hardware used by that system to support the use of capabilities are discussed.

The survey (chapter 3) is divided into three types of operating system: conven-

tional, current, and capability based. **Conventional** operating systems are typified by UNIX[1], employ monolithic kernels and are file based. The **current** operating system section describes six recent operating systems. Although these systems may use capabilities for some parts of their operation, they do not use capabilities as their only access control mechanism. The operating systems discussed in this section include representatives of distributed operation systems and micro-kernel based operating systems. The **capability based** operating system section discusses five capability based operating systems. The majority of these are based on segregated architectures; however, two password-capability based systems are discussed.

The Walnut Kernel is introduced in chapter 4.

Chapter 5 describes the Walnut Kernel from both the application programmer and the user perspectives. The topics covered include: a description of the roles of volumes, objects and capabilities, the layout of a process's address space, interprocess communication, and controlling process scheduling. A detailed list of kernel calls and their parameters is contained in Appendix A.

The design of the kernel is discussed in chapter 6. This chapter identifies the design principles that motivated design decisions, describes the partitioning of the kernel into functional components, and identifies key design decisions and their impacts. The architecture of the system is discussed in detail covering both functional decomposition and data structures. The subprocess mechanism, an innovation not present in the Password-Capability System, is described in that chapter and the motivation for its inclusion identified.

The Walnut Kernel currently exists in two forms. One version of the kernel is hosted by a conventional operating system, while the second interacts directly with hardware. Chapter 7 discusses the two versions and mechanisms for loading programs into a system without a host operating system.

A series of measurements was taken on equivalent hardware platforms for the Walnut Kernel and a BSD 4.3 based UNIX. These measurements are used to compare the speed of operations common to both operating systems. Chapter 8 provides a guide to the performance of the Walnut Kernel compared to a mature operating

[1]UNIX is a trademark of X/Open

system. It is expected that the relative performance of the Walnut Kernel will improve after optimisations are investigated and implemented. The control - a BSD 4.3 based UNIX - was optimized by its implementors.

The techniques employed in writing programs for the environment provided by the kernel are examined in chapter 9. The implementation of shared libraries and access to legacy code from UNIX systems are also discussed. Four examples of user level programs are presented to complete the picture of the environment.

A number of features that affect the security of the system has been introduced into the design of the Walnut Kernel. Examples of departures from the Password-Capability System introduced into the Walnut Kernel include the ability to derive capabilities with known passwords and the **restrict** system call[2]. Chapter 10 describes the features which have the potential to affect the security of the system, and provides an analysis of the effect of those changes.

The Secure RISC Architecture project involves elements of hardware and software. Chapter 11 describes the hardware mechanisms devised by Dr Ronald Pose and the author. This work forms the basis of a design for a scalable multiprocessor with a distributed switch. Use of deep FIFO buffers to enhance throughput and to allow each board in the system to have its own clock generator enhances the redundancy and reliability of the design.

Chapter 12 describes work being performed by others based on the concepts outlined in this document. It covers continuing work in the area of software running under the Walnut Kernel, enhancements to the kernel and the development of hardware based on the concepts outlined in 11.

In the conclusion, chapter 13, the Walnut Kernel and the hardware proposed to support it are discussed in the context of the Password-Capability System (chapter 2) and the existing systems described in chapter 3. Arguments are summarised and conclusions are drawn in relation to the success of the work as a whole.

Appendix A is a copy of *The Walnut Kernel: User Level Programmer's Guide* by Maurice Castro [Cas95]. This document provides a detailed description of the environment provided by the kernel to programmers.

[2]This system call allows rights to be removed from a capability after the capability has been created

A formal description of the **restrict** operation is given in appendix B. The formal description supplements the informal description in chapter 10.

Appendix C is a copy of *The Monash Secure RISC Multiprocessor: Multiple Processors Without a Global Clock* by Maurice Castro and Ronald Pose [CP94]. The paper outlines the concepts proposed for the hardware to be used in the Secure RISC Architecture project.

A glossary of terms used in this work is found in appendix D.

The remainder of this chapter introduces briefly the concept of capabilities and the implementations available, the concept of a persistent system, and clarifies the definition of 'threads'.

1.2 Capabilities

Capabilities provide a uniform mechanism for controlling access to, and the protection of, resources [DVH66]. The extension of the capability mechanism to allow the viewing of a capability as a form of address [Fab74] allows all system resources to be modeled as memory objects.

There are several major types of capability based systems. These systems are identified by the mechanisms they employ to prevent forgery of a capability. The architectures are:

- Tagged Architecture - employs tag bits on memory locations to identify a capability as a special object within a collection of data. The hardware prevents unprivileged code from altering the capability.

- Segregated Architecture - separates capabilities from ordinary data. Capabilities are placed in a page or segment that cannot be modified by unprivileged code. These groups of capabilities are often known as *capability lists* or *C-lists*.

- Encrypted Capability Architecture - calculates a form of checksum for the object identifier and rights conveyed by the capability known as a *signature* [GL79]. The capability and the signature are encrypted with a secret key known only to the kernel. This is given to the user process as an *encrypted*

capability. When an *encrypted capability* is presented, it is decrypted by the kernel, and the *signature* is recalculated. The capability is valid if the recalculated *signature* matches the *signature* sent in the *encrypted capability*. The test allows the system to detect attempts to forge capabilities.

- Password-Capability Architecture - employs a sparse address space. The password component of the capability makes the number of valid capabilities small relative to the total number of possible capabilities. By ensuring that the capabilities are randomly spread throughout the address space a large number of attempts at guessing a capability is required to ensure that a valid capability is found. The password-capability mechanism is statistically secure.

The Walnut Kernel implements a password-capability architecture. The architecture has several major advantages over the alternatives.

The major disadvantage of the tagged architecture is that it requires specialised hardware with additional memory bits present which cannot be used for the storage of general data which makes the architecture undesirable for both economic and portability reasons. The presence of additional specialised memory adds to the cost of the system. The need for specialised hardware restricts the choice of system.

Segregated architectures can be implemented on conventional hardware but require every operation on a capability to be mediated by the kernel. Typically, programs are required to use handles to capabilities to refer to capabilities, and to call the kernel with the handle as a parameter for all operations relating to a capability. All operations on capabilities are therefore subject to the overhead of a switch between user and kernel address space.

The security of an encrypted key system is vested in the security of the encryption algorithm and the key used by the system. Discovery of both of these items would allow the user to generate capabilities avoiding the controls imposed by the kernel and thereby compromising the system's security.

Password-Capabilities allow the use of conventional hardware without incurring the overheads present in segregated architectures, and with the flexibility of storing capabilities as ordinary data. The scheme is immune from the total compromise of security to which encrypted key systems are subject as there is no algorithm used to

determine the passwords of capability. Complete compromise of the system would require each object's master capability to be guessed.

1.3 Persistent Systems

Conventional computer systems normally have two distinct views of data; short term data or long term data. Short term data is typically stored in RAM or virtual memory and exists only for the period of time that a program is active. Long term data is often stored in the file-system of a conventional machine. Data stored in a file-system is independent from the program which created it and exists until it is explicitly destroyed. A persistent system eliminates the distinction between long term data and short term data. All data persists until it is explicitly destroyed. Furthermore, the data is manipulated with the tools used to manipulate short term data on conventional systems. These tools tend to be more flexible than the limited range of file operations usually available.

Persistent systems are designed to support persistent programming which offers a number of advantages over conventional programming [ABC+83, CAC84]. Persistent programming allows the elimination of the large component of conventional programs which is solely concerned with transforming data from file-system representation into the volatile representation that is manipulated by the program and back again. Elimination of this code saves both time and space; furthermore, it eliminates the need to perform the transformation which may distract the programmer from the primary task of the program.

Persistent systems are more attractive than conventional systems in that they provide a uniform environment for all data. The uniformity of the model simplifies the programming environment thereby easing understanding.

1.4 Threads

The *thread of control* or *thread*[3] is a path of execution through an address space. The majority of traditional operating systems permit only a single thread of control in the address space of a process. However, having multiple threads sharing an address space in some cases can be advantageous [Tan92]. If the threads of a process share the virtual address space of the process, a thread of a process has no protection from the actions of the other threads of the process.

Threads are typically used in applications with IO-bound components. The writer of the application can place the IO-bound operations in a separate thread which uses blocking IO operations. The IO-bound threads progress until they block, then other threads of the process make use of the remainder of the time-slice. Thus the process can make more efficient use of the allocated time. Without threads, a single blocked action would halt the progress of the application necessitating the surrendering of the remainder of the time-slice.

On uniprocessor systems, threads are scheduled in a time sharing manner within a process's time-slice. Each thread of a process is allocated time on the processor. Ideally, on a multiprocessor system, the threads of a process would execute concurrently; however, a number of systems claim to support threads, but support only the time-slice semantics supported on uniprocessors.

Threads may be implemented at either the user [BS90] or the kernel levels. User level packages typically make use of kernel upcalls or a yield call built into the package to allow switches between threads. On systems which support upcalls, an upcall is made when the kernel detects an event which would cause a process to block. The upcall invokes a management routine which either selects another thread of the process to run or surrenders the remainder of the time-slice. The yield call is used to switch between co-operating threads by indicating that the current thread is ready to allow another thread to proceed. Kernel level thread packages make use of primitives that are implemented in the kernel.

[3] Also referred to as a lightweight process [Mic90], however, this term promotes confusion. The term lightweight process was originally used to distinguish UNIX style processes from Multics style processes

The Walnut Kernel does not provide support for the concurrent execution of multiple threads of execution within a process on multiprocessors. A mechanism with time-sliced semantics is provided to let user processes handle asynchronous events.

The absence of concurrently executing threads is not regarded as a disadvantage. Walnut Kernel processes can share address spaces in a more flexible and controlled manner than concurrently executing threads, and hence they provide similar functionality to concurrently executing threads, but with the advantage of greater control of access to memory.

The design of the Walnut Kernel recognised that the need for threads is driven by two factors: the high cost of creating a process and the difficulty of sharing address space between processes. The first motivation for the use of threads is dealt with in two ways by the design of the Walnut Kernel. The design of the kernel endeavors to keep the cost of generating a new process to a minimum, and also, as processes are persistent it is practical to maintain processes as servers which perform tasks on demand avoiding the cost of generating a new process each time a function is required. The second motivation for threads is addressed by observing that the basic model of the system (capabilities allowing access to objects) is based on the interprocess sharing of code and data through memory objects. The shared memory model contrasts with file based systems which share information through files and pipes necessitating the use of file protocols. File protocols perform well for sharing data between processes when data structures which map well onto streams are used. However, more complex data structures (such as trees and graphs) are generally less well handled.

Chapter 2

A Password-Capability System

This chapter describes the 'Password-Capability System' developed in the Department of Computer Science, Monash University, circa 1985 [APW86, And87, Pos91, APW85, AW85], which represents the historical background of the current project.

The 'Monash Multiprocessor Project' had both hardware and software components. The hardware developed for the project consisted of a collection of processor and memory boards on a shared high speed bus. The processor boards provided a set of capability registers to implement the address transformations required. To the best of our knowledge this system [APW85] introduced the concepts of *password-capabilities* and money based garbage collection. Subsequently a number of systems adopted the password-capability mechanism (see chapter 3).

2.1 The Kernel

User processes operated in a uniform virtual memory. The address space was divided into volumes each of which contained a number of objects. Volumes were permanently associated with pieces of hardware. The majority of volumes were associated with storage media, such as disk packs. Some volumes were associated with a particular multiprocessor. Typically, the latter type of volume was used to access physical devices attached to that multiprocessor. When an object was created it was permanently associated with a single volume and allocated a serial number on that volume. Serial numbers were required to identify uniquely a given object on a

Volume	Serial	Password 1	Password 2

Figure 2.1: A Capability in the Password-Capability System

volume for all time and were never reused. Serial numbers were allocated consecutively on a volume. The volume and serial numbers of objects were defined as 32-bit quantities.

All objects in the system were composed of contiguous sections of the virtual memory. Processes gained access to the contents of an object by 'mapping in' part of the memory object into the address space of the process. After mapping bytes X to Y of an object into addresses A to B of a process's address space, reference to address $A+\epsilon$ will reference byte $X+\epsilon$ of the object, or a word of bytes starting with this byte. 'Mapping in' is not copying; If a process writes to address $A+\epsilon$, byte $X+\epsilon$ is altered, and the byte will seen to be altered by any other process which has the byte mapped in.

The Monash Multiprocessor Project introduced a mechanism for providing non-segregated capabilities employing probable security. This mechanism differed from the alternatives of tagged architectures and segregated architectures which both distinguished capabilities from other forms of data. Capabilities were 128-bits in length and had 4 components (see figure 2.1). The *volume* and *serial* number formed the name of the object. *Password 1* and *Password 2* uniquely identified a capability for the named object. Passwords were allocated purely randomly; there was no encoding of access rights in the password. The security of the mechanism was derived from the small number of valid passwords in comparison to the total number of passwords. The password-capability acted as an identifier for a set of access rights to an object. Many capabilities could convey the same access rights to an object but with differing passwords. A table of valid capabilities and their associated rights was stored on the volume containing the object to which the capabilities referred. This table was known as the 'catalog' and was accessible only to the kernel. Capabilities were revoked by deleting the entry for the set of rights associated with the capability.

The password-capability scheme had a number of advantages over the alternatives of tagged architectures and segregated architectures. It was not subject to the

major disadvantages of tagged architectures. The first disadvantage was economic as tagged architectures must supply physical memory for tag bits, which contribute to the cost of the memory but are unavailable as general purpose memory. The second problem was that the processor must support tags. Special instructions must be used to access capabilities safely within a tagged architecture requiring hardware support, and hence limiting the choice of processor. In addition, as password-capabilities were stored as ordinary data, the mechanism was not subject to the limitations of segregated systems. Segregated systems required the kernel to provide mechanisms to move, copy and communicate capabilities, preventing capabilities being treated as ordinary data which made it difficult to pass capabilities to processes or users outside the computer system.

When an object was created the kernel returned a single capability to the creator. This capability was known as the master capability. It described the complete object and held all possible access rights to the object. Other capabilities, with restricted rights, could be derived from the master capability or from its children. The capabilities for an object were logically organised in a tree structure. The master capability was the root with derived capabilities being the other nodes of the tree. Each internal node of the tree is the root of a subtree which has rights equal to or weaker than the root of the subtree. Deletion of any capability results in the deletion of its descendants. Deletion of the master capability results in the destruction of the object as no further access to the object is possible.

Any process that held a capability was capable of using it to access the object referred to by the capability reducing the need to create many capabilities with equivalent rights.

The Password-Capability system did not support the concepts of ownership or dependency between objects. Once an object was brought into existence any process knowing a capability could make use of it. Capabilities were copied, passed between processes and stored by processes as ordinary data[1]. In addition, capabilities could be held outside the system, by a peripheral or a user, and still refer to a valid object making it impossible to determine if there were any entities which knew a valid

[1]The system allowed capabilities to be stored using any representation. Thus an encrypted capability was as valid as a plain text capability

capability for an object. These properties prevented the use of existing mechanisms for garbage collection which depended on ownership, reference counts, or reference tracing. Two mechanisms were used to implement garbage collection. An object was destroyed when no valid capabilities for that object could exist, that is, when the master capability for the object was deleted. The second mechanism relied on a simple charging scheme. Each object had associated with it a store of money from which it periodically paid rent. Bankrupt objects were deemed to be garbage, and were destroyed.

The money mechanism was generalized to form an economy within the Password-Capability System. Both ordinary objects and processes had quantities of money associated with them. Money was managed directly by the kernel and was distinct from normal data items. Money could be transferred between objects and was consumed when services were provided by the kernel. Each capability had a monetary value associated with it. The money associated with the master capability represented the total store of money held by the object. Derived capabilities were associated with a value that represented the amount of money that could be withdrawn through that capability. A withdrawal or deposit caused the values held by each of the ancestors of the capability to be updated by the amount of the withdrawal or deposit. A withdrawal would occur only if all the values held by the capability and its ancestors would be non-negative after the operation was performed. The money mechanism allowed processes to charge for services to cover the costs of performing work and allowed for the possibility of charging for the use of a program unlike the conventional scheme of paying a license fee for access to a program which may or may not be used.

Processes had no internal parallelism and hence could run, at most, on one processor at a time. However, processes could be moved between processors. This allowed the Password-Capability System to schedule processes on the first suitable processor that became available.

Lockwords were used in the Password-Capability System to prevent non-trusted code from distributing information shared by that code. The confinement mechanism identified two classes of capabilities: alter capabilities which could be used

to convey information to third parties and non-alter capabilities which were strictly read-only. The top bit of the password field was set to indicate that a capability was an alter capability. The remaining 63-bits of the password were chosen randomly. Each process had a 63-bit lockword associated with it. The lockword could not be read. Locking the process caused the lockword (L') to be set to the exclusive-or of the original lockword (L) and the argument of the operation which sets the lockword (V):

$$L' = L \oplus V$$

Passwords were not encrypted in processes with a zero lockword. To execute an untrusted package, another process (P_2) was created with a non-zero lockword (L_2). This lockword is generated from the creating process's (P_1) lockword (L_1) and an arbitrary value (V_L) selected by the creating process.

$$L_2 = L_1 \oplus V_L$$

The passwords of alter capabilities (C_{pass}) in the new process (P_2) are encrypted using the process's lockword before being used by the kernel.

$$C'_{pass} = C_{pass} \oplus L_2$$

Non-alter capabilities are not affected by the value of the lockword. Provided process P_1 was not locked (ie. had a zero lockword) this mechanism allowed process P_1 to pass capabilities to process P_2 by encrypting the capabilities passed using V_L. The mechanism, however, prevented P_2 from passing on those capabilities as the value of L_2 was unknown to P_2. Any capability created by a process was encrypted using the process's lockword. Process P_2 could create objects for its own use or for the use of any process which knew L_2. Program code and data was made available without the risk of data leakage by using non-alter capabilities to map in read-only code and data. Groups of locked cooperating processes could be created by using the same lockword for each of the processes when they were created.

The Password-Capability System allowed capabilities to confer the following rights:

Money Rights - Three separate rights were supported to allow access to information on money. The *drawing* right determined the maximum amount of

money that could be withdrawn through the capability. This value was reduced by withdrawals and increased by deposits. Possession of a *balance* right was necessary to determine the amount of money that could be withdrawn via a capability. The result returned was the minimum of the drawing right of the capability and its ancestors' drawing rights. A *deposit* right allowed money to be added to the drawing right. A separate right was required to provide control over a potential covert channel which functioned by varying the drawing right.

Window Rights - These rights allowed the visible region of the capability to be specified. A window consisted of a set of consecutive words within an object starting on a word boundary (*offset*) and extending an integral number of words (*size*). Derived capabilities were required to have windows which were ranges of the capability from which they were derived. Associated with each window were the access rights: *read*, *write*, and *execute*.

Process Rights - Processes had four additional rights in addition to the rights held by other objects. A *message* right allowed the holder to send a 16-word message of arbitrary content to a process. If the process was waiting on a message, the process was awakened. The *suspend* right allowed the holder to suspend and resume the process. The internal state of the process could be determined by holders of a capability with a *status* right. Holders of a capability with a *condition* right could initialize a suspended process. This allowed the partial modification of the state of a process.

Suicide Right - Possession of a *suicide* right allowed a capability to be used to destroy itself. This allowed the distribution of the same capability to antagonistic processes but prevented one party depriving the others of the use of the capability. Suicide right was unique in that capabilities with suicide right could be derived from capabilities without suicide right.

2.2 The Hardware

The kernel of the Password-Capability System relied heavily on many of the features of the hardware designed to support the system. Most notable of these features is the presence of segment registers which map capabilities onto a paged virtual memory. The presence of specialised hardware resulted in significant simplifications in the design of the operating system.

The prototype hardware was based on the NS32032 processors operating at 10MHz. The NS32081 was used to provide floating point support. Memory management was provided through the custom hardware described in this section. The bus connecting the processor boards operated at 40MHz.

Figure 2.2 illustrates the overall structure of the hardware. The system employed a number of processor boards which communicated with a number of shared memory boards over a single high speed bus. Each processor board supported a set of capability registers (also known as *window registers*) which were used to transform the addresses generated by the processor into addresses in the *Intermediate Address Space* (IAS). The IAS addresses were then used to access data via the bus. Each memory board performed a check to determine if the address required was present in the pages stored on the board and allowed access if the page was present.

The hardware on each processor board provided logical address to IAS address translation (see figure 2.3). The logical address of the processor was composed of 3 parts: a window register identifier, an offset into the window and a byte offset into a word. The top 5 bits of the logical address were used to identify the appropriate window register from the 32 supported. The *offset* and access mode were checked against the *window size* and *window rights* respectively. If the access was legal, the sum of the *IAS Base Address* and the offset into the window was calculated and used as the IAS Address of the access.

The hardware supported checking of read, write and execute access rights and limit checking to word granularity.

On receiving an IAS address, a memory board checked its hash table to determine if the page required was stored on it (see figure 2.4). The check was performed by using the middle 12 bits of the IAS address as an index into the hash table. If the

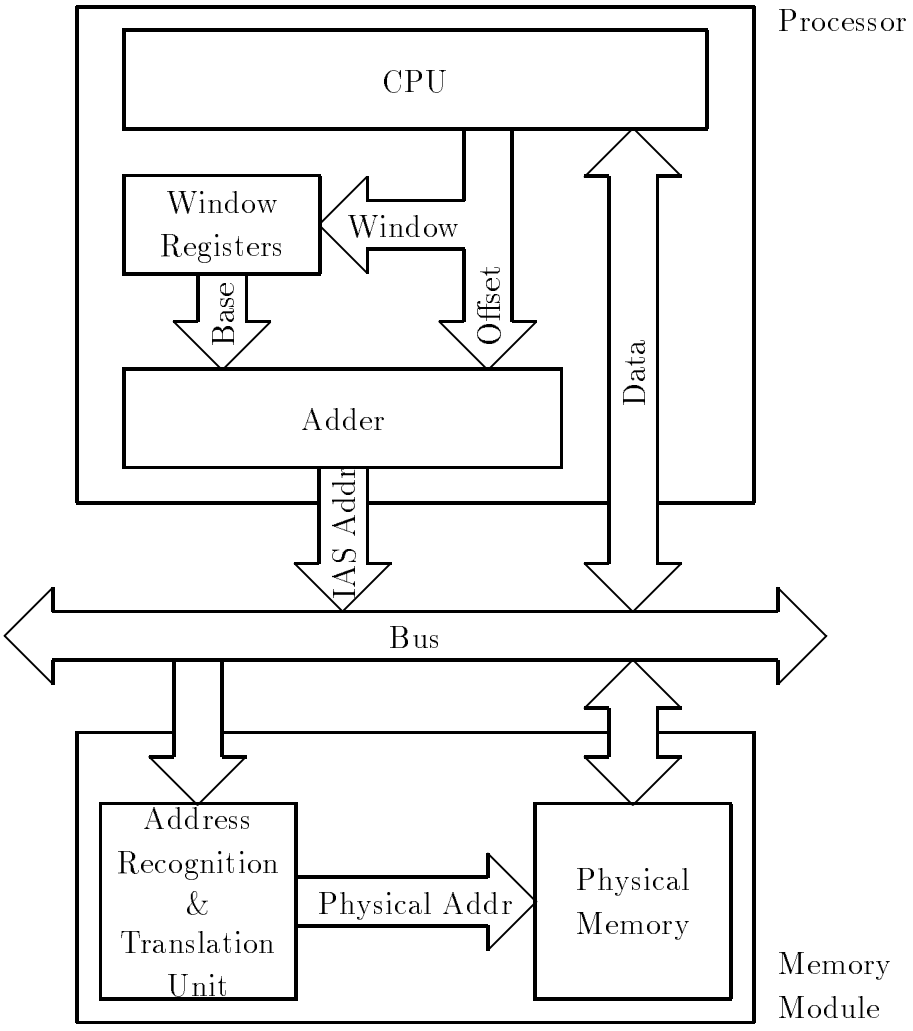


Figure 2.2: Block Diagram of the Password-Capability System's Hardware

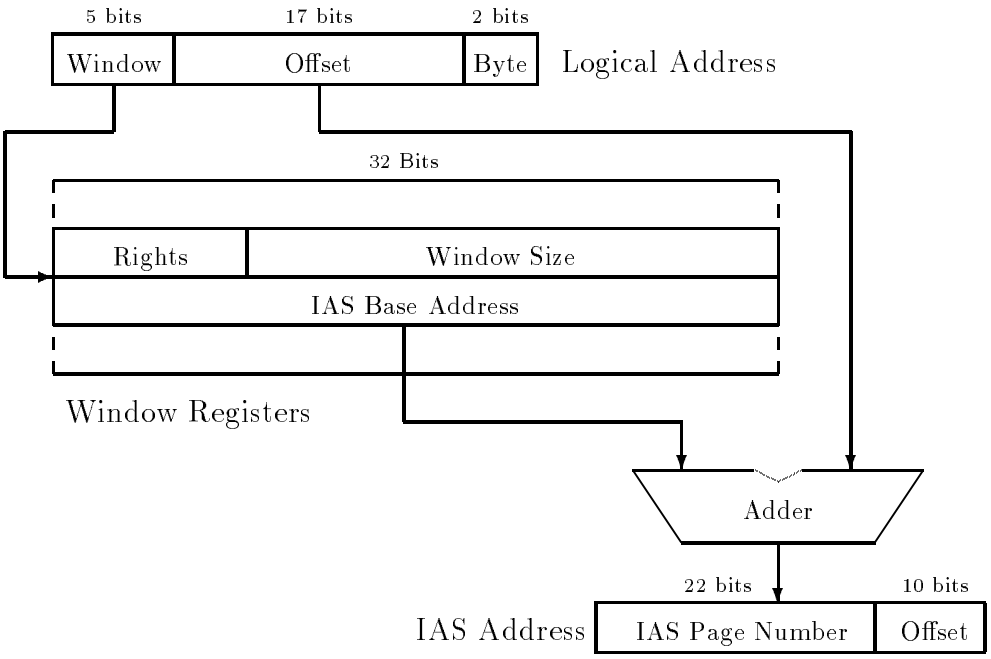


Figure 2.3: Logical Address to IAS Translation

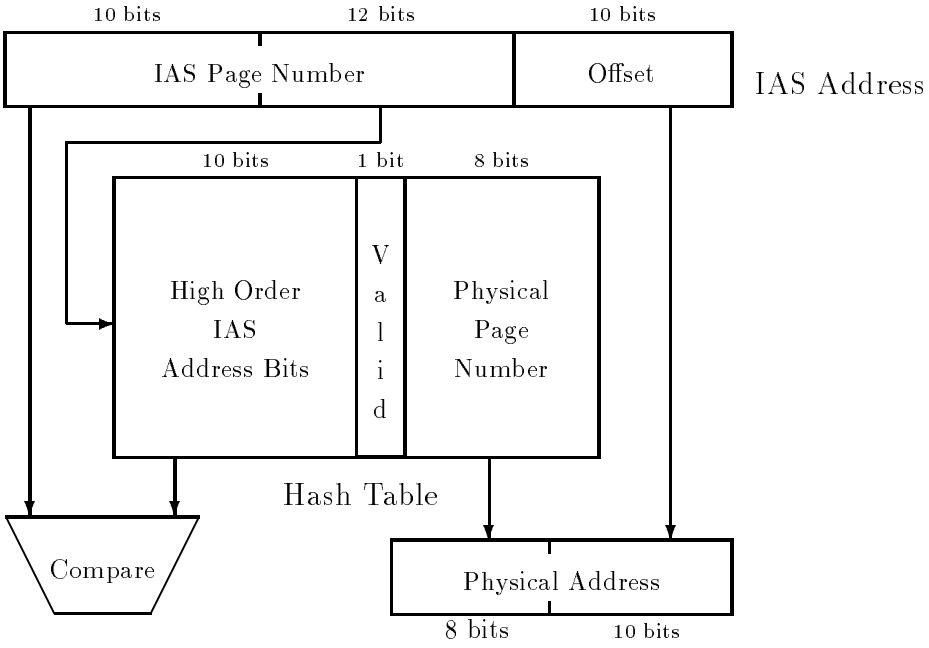


Figure 2.4: IAS Address to Physical Address Translation

10 bits stored in the hash table matched with the 10 high order bits from the IAS address and the page was marked valid then the page was stored on the memory board. If a match occurred, then the required *physical address* was generated by concatenating the *physical page number* and the offset into the IAS page. The physical address was used to gain access to the appropriate item in memory.

Multiple memory modules which recognized the addresses of IAS pages stored within them allowed the near arbitrary[2] allocation of pages to memory modules. This property allowed the distribution of pages across memory modules to equalize the load across the modules.

As the physical memory of the machine was limited, objects could not be retained in the IAS indefinitely. The large size of the IAS allowed a simple management strategy to be employed: Objects were retained in the IAS until the space became exhausted. At that point all objects were flushed from the IAS, window registers were marked invalid and caches were flushed. Subsequent accesses bring objects back into the IAS.

The Password-Capability System's operating system was strongly supported by the system hardware which provided direct support for capabilities through purpose built capability registers and a number of intelligent peripherals. The peripherals included shared memory with VAX 11/750 and purpose built IO controllers.

[2]The use of a hash table instead of a content addressable memory required that two pages which hashed to the same value cannot be stored on the same memory module

Chapter 3

Survey

This chapter surveys a number of operating systems to provide a historical context for the work on the Walnut Kernel discussed in later chapters. The chapter comprises several sections. The first section (section 3.1) provides an overview of a conventional system. UNIX is selected for its ubiquity among current commercial systems. The second section (section 3.2) examines several current operating systems which display either a distributed or micro-kernel architecture. Micro-kernel architectures are significant in that they bring software engineering practices to the kernel level while distributed architectures aim to provide scalable mechanisms for implementing operating systems on systems with more than one processor. A notable absence from this section is Microsoft's Windows NT[1] operating system. Although the system is likely to be of great commercial importance, the information available [Cus93] indicates that from the perspective of innovation within the kernel, the operating system is similar to the other micro-kernel based architectures surveyed. The third section (section 3.3) covers a number of capability based operating systems. The operating systems covered in earlier sections may make use of capabilities in a limited role, however, those in this section use capabilities for all access and naming functions. The final section (section 3.4) draws conclusions about trends in the development of operating systems.

The survey uses Tanenbaum's [Tan87] four major components of an operating system (process management, input/output, memory management and file system)

[1] Windows NT is a trademark of Microsoft Corporation

to provide a basis for comparing the systems presented.

3.1 Conventional Operating Systems

The UNIX family of operating systems is based on the work of Ritchie and Thompson [RT74]. This brief overview covers the gross features of UNIX and some of their implications. UNIX has been widely used in both commercial and research environments. The interest of the research community has prompted many detailed studies of the characteristics of the operating system. This survey identifies some key features of the operating system that are either typical of conventional operating systems or have had significant influence on the development of other operating systems. The features discussed are: the style of processes provided, protection and the implementation of the kernel.

In direct contrast to Multics and VMS where, due to the significant costs involved in creation and destruction, processes are reused, the UNIX system employs inexpensive processes[JJ91c]. Low cost processes[2] allow the application of more than one process to the task of solving a problem. A problem can therefore be decomposed into a number of smaller tasks, with the possibility of using utilities for some of those tasks. The use of small utilities confers the software engineering advantages of encouraging both the reuse of code and providing encapsulation of code. Mechanisms which allow the easy reuse of code make it desirable to invest time in the demonstration of correctness and optimisation. The high cost of Multics processes encourages the use of an *in-process* model of protection[Kee79, Tan92, Sal74]. Under the *in-process* model, encapsulation occurred at the subroutine or module level. The low cost of UNIX processes and the absence of hardware support for more than two privilege levels favors the *out-of-process* protection mechanism adopted by UNIX.

Protection on UNIX systems is a combination of two mechanisms. The primary mechanism provides two classes of user: ordinary users who have access to a limited selection of kernel operations, and super users, who have access to all functions provided by the kernel. The functions of the file system comprise the second mech-

[2]The term 'lightweight process' has not been employed to avoid confusion as the term has been more recently acquired by SUN and other vendors to describe their thread implementations

anism. The UNIX file system associates with each file an owner, a group, and a set of permission bits. The permission bits determine the level of access to the file that the owner, the group and all other users are allowed to have. In addition to this, executable files have the ability to inherit the powers of the owner (setuid) or group (setgid) of the file for the duration of the execution of the program. The presence of an omnipotent user - the super user - and the ability to inherit the powers of the owner of a process, allow privileged functions to be made available to users in a controlled way.

There are two major families of the UNIX kernels in current use: those derived from the commercial System V[GC94] and those derived from the research operating system created by the CSRG at Berkeley[LMKQ90, JJ91a]. Although there are major differences between the systems - for example BSD places the kernel address space at the top of the process address space[JJ91b] in contrast to System V which places it at the bottom of the address space - the systems are sufficiently similar at the conceptual level that no further distinction will be made between these versions of the operating system in this chapter[3].

The UNIX kernel is *monolithic* in that it is constructed as a single program existing in a single address space. The kernel contains code to support virtual memory, user requested system functions, devices and the file system. A number of consequences flow from this method of organising the kernel, the most significant of these is the absence of protection of the kernel from the actions of device drivers, and the requirement that the system needs to be rebuilt and restarted to incorporate a new device driver or an alteration to an existing device driver.

The kernel manages access to the hardware through the device abstraction. A device appears as a special file within the file system. File operations and I/O control operations (ioctls) are applied to the special file and translated by the kernel into operations on the device.

The UNIX system introduced a number of significant features to mainstream operating systems while retaining the common monolithic implementation of the kernel. Development has continued. The most significant change has occurred with

[3]The various 'kernelized' and 'emulated' versions supported over micro-kernels are excluded from this discussion

the introduction of 'threads' by a number of vendors.

UNIX displays the conventional operating system characteristics of transitory processes which use the file system to provide persistent storage. The use of special-files as interfaces to input/output devices provides greater uniformity[4] within the system than in other conventional systems which have separate mechanisms for dealing with devices. Finally, in a feature typical of the majority of conventional systems, most versions of UNIX currently provide only limited support for memory to be shared between processes.

3.2 Current Operating Systems

This section describes a number of recently developed operating systems which exhibit the features of either distributed operating systems or a micro-kernel architecture.

The advent of powerful low cost microprocessors has altered the economics of the provision of computer resources. In general, it is now more cost efficient to use a large number of processors than to use a single large processor to provide a given amount of computational power. The shift in the cost of provision of service has been the major motive for the development of distributed operating systems.

Distributed operating systems allow many users to work on a collection of processors linked by a high speed network. Tanenbaum [Tan92] identifies the following advantages and disadvantages of distributed systems:
Advantages:

- Economics - Microprocessors offer a better price/performance ratio than mainframes

- Speed - Distributed systems are not subject to the fabrication and construction requirements of single processor systems. Distributed systems can be constructed with more total computing power than a mainframe constructed with similar technology.

[4]The *Input Output Control* (ioctl) mechanism which allows operations which are not defined for files to be performed on devices partly negates this advantage

- Inherent distribution - Some applications involve spatially separated devices. This class of problems has a natural or inherent decomposition onto a distributed architecture.

- Reliability - The loss of a single CPU or group of CPUs reduces the total performance of the system but should not prevent the system from operating. In a centralized system the loss of a single component, typically, results in the failure of the complete system.

- Incremental growth - As distributed systems are composed of computational units on a high speed network, computing power can be added in these units or multiples of them allowing the power of the system to be scaled up gradually.

Disadvantages:

- Software - The design and implementation of distributed programs is significantly different from sequential programs. As a result, little software exists for distributed systems

- Networking - Distributed systems rely on networks between processors to allow for the transfer of information, and the co-ordination of operations between processors operating on a problem. These networks can saturate or suffer from other problems.

- Security - It is necessary for a distributed system to promote easy sharing and access to data among the processors of the system to allow the processors to work co-operatively on that data. Easy access to data applies to secret information as well.

Traditional monolithic kernels provide a large number of system functions which are used directly by user programs. This contrasts with the micro-kernel approach of providing a small number of essential services from within the kernel, and employing user-level servers to provide the bulk of the services expected by user processes. Micro-kernels make for greater flexibility in the services provided, and in the implementation of those services, than do monolithic kernels. Tanenbaum [Tan92]

identifies the four minimal services[5] implemented within a micro kernel as:

1. An interprocess communication mechanism.
2. Some memory management.
3. A limited amount of low-level process management and scheduling.
4. Low-level input/output

Micro-kernels also offer advantages to the implementor of an operating system. Because of the limited functionality required from a micro-kernel, the complexity of the micro-kernel is considerably less than that of a monolithic kernel. In addition, micro-kernels allow software engineering techniques to be applied to operating systems. By dividing operating system services into a number of user-level servers, services can be encapsulated into logical units, reducing design and maintenance difficulties.

A number of operating systems introduced in this section make some use of capabilities. However, none of these operating systems is capability based as they do not use capabilities as the sole mechanism for determining access to each object and process under the control of the system.

3.2.1 Amoeba

Amoeba [Tan92, vRT92, MvRT+90] is a micro-kernel based distributed operating system supporting multiple users in a transparent environment. It was initially designed by Andrew S. Tanenbaum, Frans Kaashoek, Sape J. Mullender and Robbert van Renesse in 1981 at Vrije University in Amsterdam.

This distributed operating system runs on a network of dissimilar workstations and servers. The network is intended to contain a large number of CPUs with tens of megabytes of memory available to each CPU. Shared memory, if present, is exploited

[5] Tanenbaum identifies the provision of low-level input/output as a necessary service of a micro-kernel. As there are a number of micro-kernels which delegate IO operations to user level servers it is clear that this statement requires modification. A more appropriate statement would be 'Management of access to low-level IO'

to improve message passing performance. The network has four classes of nodes: workstations, pool processors, specialised servers, and wide area gateways. There is one workstation allocated to each interactive user and it runs tasks which require fast interactive response. These tasks include the window manager and text editors. Pool processors consist of one or more CPUs which are allocated, as required, to a task. At the completion of the task the processor is returned to the pool for others to use. A number of specialised servers is present to perform functions which either need to run on a separate processor or need to be in continuous operation to provide greater efficiency. Typical specialised servers would include: directory, file and block servers, and data-base servers. Finally, wide area gateways link Amoeba sites into a seamless system.

Each machine in an Amoeba system runs a functionally identical micro-kernel. The Amoeba micro-kernel manages processes and threads, supports low-level memory management, provides transparent communication between threads, and handles I/O.

Processes define an address space which may be shared by a number of threads of execution. Each thread has its own register set, stack and program counter. To simplify the provision of blocking I/O the threads are scheduled by the micro-kernel.

The micro-kernel provides memory management services which allocate and deallocate blocks of memory, known as segments. No limitations are placed on how a process uses a segment, and segments may be mapped into and out of a process's address space, as necessary. When a segment is mapped out of a process's address space a capability for that segment is returned. This capability may be passed to other processes to allow them to load the segment into their address space. A segment remains in the system memory even when it is not mapped into a process's address space.

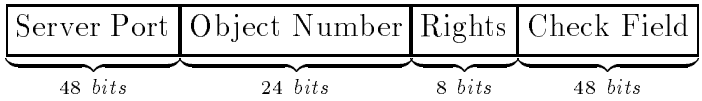
A client server model of distributed processing is implemented through the use of Remote Procedure Calls (RPCs). This mechanism is used to provide transparent communication between threads. The Amoeba RPC mechanism consists of 3 phases:

- `do_operation` - send a message to the server and block this thread of execution until a reply has been received. This call is issued by a client requesting a

service. The destination thread is identified by a 48 bit number known as a port.

- `get_request` - announce that the current thread is willing to receive a message sent to a nominated port. This call is used by a server to advertise the server procedure attached to a specified port.

- `send_reply` - send reply back. This call is used by the server to reply to the client requesting the service. On receipt of this message the client thread is unblocked.



Server Port - identifies thread that manages object

Object Number - uniquely identifies object managed by server

Rights - identifies operations permitted by holder of capability

Check Field - protects capability against forging or tampering

Figure 3.1: Components of an Amoeba Capability

Although Amoeba uses capabilities (figure 3.1) with cryptographic check fields to identify objects within the system, Tanenbaum *et al* do not identify Amoeba as a capability based operating system. The use of capabilities within the Amoeba system is not uniform in that all the fields of a capability are used and validated to load or access an object, but only the server port field of the capability is validated for RPC operations. The other fields are passed to the server. The server may interpret the other fields without restriction.

A sparse port name space (a 48 bit port number is required to access a port) partly protects servers from unwanted access attempts. Only clients of a server are informed of the server's port numbers, and hence only valid clients of a server should be able to access the server. If the port number is leaked, additional mechanisms

are required to ensure that a server is not abused. This protection may be provided within the server as it may reject messages which are not of the correct form.

To protect against intruders emulating servers, Amoeba uses a one-way function to encrypt port names. Clients of a service are given the port name P which is derived from G, the secret port name known to the server using the one-way function. Servers use their secret port name to advertise their service, G is transformed by either hardware or software (an F-box) into P, and the service is made available from a host to the network. Clients request a service using the publicly known port name P, and are connected with the server. All port names advertised by servers are transformed by an F-box, and as G is not publicly known, intruders cannot emulate a server.

Other services provided on Amoeba systems are not supplied by the micro-kernel; instead they are made available by user level servers. The most critical of these services are the directory and file system services.

The basic file system service provided with an Amoeba system is known as the Bullet service. The Bullet server creates and manages immutable files. As the files cannot be altered after creation, the size of the file is known at the time of access. This feature is exploited to allow files to be stored contiguously on disk and in the main memory cache. As the files are stored in contiguous blocks, only a single read operation is required to recover the file from disk, and a client can read a file, completely, in a single RPC operation.

ASCII string	Group 1	Group 2	Group 3
Mail	Cap 1.all[a]	Cap 1.ro[b]	Null Cap[c]
Games	Cap 2.all	Cap 2.rw[d]	Cap 2.ro
Exams	Cap 3.all	Null Cap	Null Cap

[a] capability with all rights
[b] capability with only read right
[c] capability conferring no rights
[d] capability with read and write rights

Figure 3.2: Logical Representation of an Amoeba Directory

The directory server provides a mapping between ASCII strings and capabilities.

The directory server organizes the mappings into collections known as directories (figure 3.2). Directories can be shared with other users, and grant access to the capabilities contained within the directory in a controlled way. The directory is structured as a two dimensional table consisting of a collection of strings naming the object, and a number of capabilities which grant access to the object. The capabilities will frequently allow different levels of access to different groups. As a directory is an object within the Amoeba system, it is represented by a capability that can be placed within a directory. Directories can be used to represent an arbitrary graph structure.

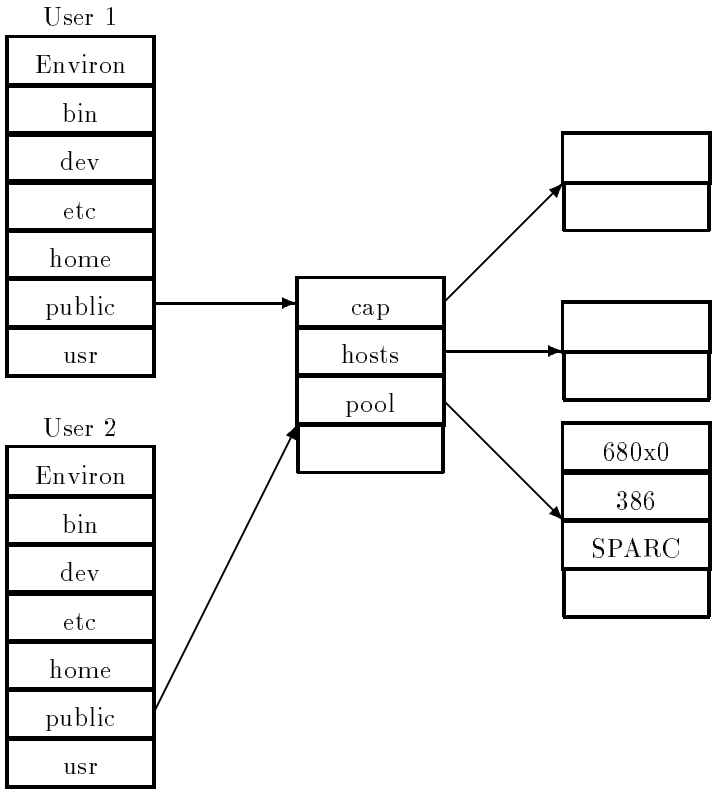


Figure 3.3: Basic Amoeba Directory Hierarchy

Each user is provided with a root directory (see figure 3.3) which contains entries for the user's environment, binaries, I/O servers, administrative information, home directory, and the public directory. The public directory contains the root of the public shared tree. The most significant entries in this area are: the *cap* directory which contains capabilities for public servers, the *hosts* directory which contains

capabilities for host specific servers, and the *pool* directory contains capabilities for pool processors.

In summary, the Amoeba system is a distributed operating system that provides transparent access to all the facilities, within the system, without the user being made aware of the location or nature of the resource. Specialised nodes can be exploited, without distorting the model, as they appear as servers within the system. Capabilities and sparse address spaces are used within the system to provide a measure of security, however, they are not exploited to their fullest extent.

3.2.2 Mach

The Mach micro-kernel [BGJ+92, RJO+89, Loe92, Tan92] forms the basis of a number of operating systems in current use including OSF/1, many research projects (including [VSK+90]), and has been suggested as the basis for a portable form of OS/2. Mach's direct ancestors were designed to demonstrate that modular operating systems based on message passing were feasible. The earliest versions of Mach were monolithic and designed to be compatible with UNIX in order to exploit software becoming available for UNIX systems. At that stage Mach provided support for multiple processors, threads and interprocess communication, and although it supported network operations, it was not envisaged as a distributed operating system. With the advent of Mach 3.0, the code derived from Berkeley UNIX was removed, and Mach was transformed into a micro-kernel supporting distributed operations. The features of the Mach 3.0 micro-kernel are examined below. For convenience, Mach is used to denote Mach 3.0 in the following text.

Mach provides the minimal services required of a micro-kernel: task management, memory management, interprocess communication and device support. In addition, both multiprocessors and multicomputers are supported in the micro-kernel, and system call redirection is provided.

Two abstractions are used for task management: tasks and threads. A task corresponds to a collection of resources, including the task's address space, and access to communication facilities to both the kernel and the server. Threads are paths of execution within a task. More than one thread may be active within a task

at a given time, and on a multiprocessor, several threads may operate concurrently. Both the control of tasks and the scheduling of threads can be mediated by user level programs. User level programs can have complete control over the scheduling of processes.

Memory management consists of mapping contiguous sections of Mach objects into the address space of Mach tasks. The micro-kernel manages the main memory as a cache of sections of Mach objects. A significant feature of Mach's memory management is that a user level page fault handler may be specified. User level code can control the fetching of pages from backing store. This feature has been exploited in the implementation of persistent systems such as the Napier88 environment [VSK+90].

Interprocess communication is provided through the use of ports. A port is a data structure accessible only to the micro-kernel which consists of a fixed length, ordered list of messages. The port data structure is used to implement all communication under Mach. When a port is created, capabilities known as *port names* are created. Associated with these capabilities are *port rights*. Three types of *port rights* exist: *receive right*, *send right* and *send-once right*. Only one task may hold a receive right for a port although many tasks may hold a send right. Ports are only destroyed when the receive right is destroyed. A task gains access to a port by loading the *port name* into its port name space. The micro-kernel maintains a count of the number of tasks that have a port loaded. The port abstraction is used to control every element of the Mach system. Operations on tasks, threads, and objects, are all performed by sending messages to the appropriate port.

Low level device I/O is modeled using the port mechanism. Messages can be sent to ports to transfer data and control devices. Mach is capable of supporting both synchronous and asynchronous devices as it separates read and write messages from request and reply messages.

To assist in the provision of the binary emulation of operating system environments, Mach incorporates system call redirection. When one of the redirectable system calls or a redirectable exception occurs, user mode code, within the calling task, can be invoked to handle the call. A library of routines emulating the system

calls of the emulated operating system can be incorporated into a task. The system calls are activated seamlessly. The redirection need only be set up once as it is preserved in child tasks after a fork operation.

Mach supports multiple processors through the use of processor sets. The members of these sets form a pool of processors which can be used to schedule tasks and threads assigned to the set. The use of processor sets provides a mechanism for the control of the scheduling of threads within a multiprocessor or a multicomputer.

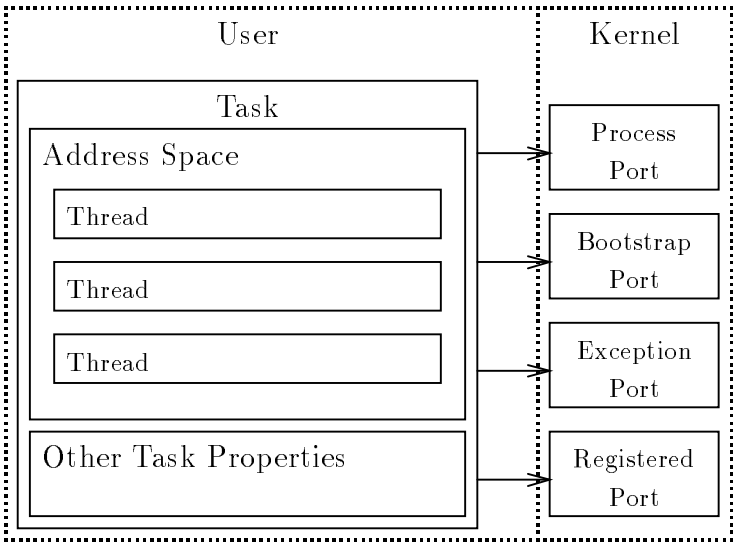


Figure 3.4: A Mach Task

Figure 3.4 depicts a Mach task. In addition to its address space a Mach task contains default values for the processor group and scheduler parameters to be used by threads operating in its address space, and a collection of statistics relevant to the history of the process. The process port is used to request services from the microkernel. The bootstrap port is read by the first process in the system to determine the names of kernel ports. The exception port is read to determine the nature of any errors. A collection of registered ports provide access to standard system services.

Each Mach thread has a port that can be used to control it. As ports are accessible to all threads within a task, these ports allow a thread to control itself and other threads within a task. Using this facility, it is possible to construct user process managed thread packages.

Emulation libraries are the key component that allows Mach to support operating system environments. Emulation libraries translate program requests into Mach operations. Efficiency is improved by placing the emulation library into the same address space as the program expecting the emulated services because this reduces the cost of communication between the emulation routines and the program. As code within the emulation library shares the address space of the programs it supports, it is unable to protect data that it accesses from the program. Thus the emulation library mechanism cannot be used to protect sensitive data from programs.

The use of emulation libraries allows several 'personalities' to exist on a Mach system at one time, executing in parallel. In addition to using Mach functions to support operating system functions, it is possible for the emulation library to use the facilities of other personalities. This feature is most commonly used to allow operating environments to use files hosted by other operating environments.

The Mach micro-kernel has demonstrated that micro-kernels can support multiple operating system interfaces within a single system. Access to process scheduling and memory management from user level processes has made Mach a popular tool for the implementation of experimental operating environments. Mach is not capability based, as within it, capabilities are used only in a limited way to protect access to ports.

3.2.3 Plan 9

Plan 9 [PPTT92] is a distributed multi-user operating system which bears a strong resemblance to UNIX at the user interface level. Plan 9 draws on concepts found in UNIX, generalized for the new environment, rather than attempting to implement previously untried concepts. The operating system is aimed at a similar environment to the Amoeba system (see section 3.2.1), in that it is composed of workstations (known as terminals), pool processors (known as CPU servers) and file servers. Resource sharing, and reducing administration were priorities in the design of the system.

By using a limited number of powerful abstractions it is possible to produce a small kernel that performs the functions of larger kernels. The micro-kernel philos-

ophy of minimising functionality of the kernel, and placing maximal functionality into user processes differs from Plan 9's philosophy of minimalism and uniformity. Unlike micro-kernels, Plan 9 incorporates elements of the file system into the kernel. The file system serves as the major abstraction of the operating system.

Resource sharing is promoted by using identical kernels on each terminal and CPU server. This allows users to choose whether to run programs on their terminal or on a CPU server. The distribution of tasks varies with the bandwidth of the link between the terminal and CPU servers. On slow links users tend to place programs so as to reduce the communications cost. On faster links users tend to run programs locally, unless the program is a data or compute intensive job.

All the resources of a Plan 9 system, other than program memory, are represented as files within the file system. The strict tree structure imposed on the file system (links are not supported) and the presence of all program accessible resources makes the file system into a uniform name space. Physical devices, abstractions[6] and software concepts[7] are all representable by file systems. File systems can be implemented within the kernel as a driver, as a user level process or as a remote server. Access to file systems outside the kernel is performed through the kernel's *mount driver*. The mount driver converts operations into request messages which are relayed to either the user program or the remote server which implements the file system functions.

The uniform name space, and the ability to implement file systems in either the kernel or user code, provides a uniform mechanism to access either kernel or user functions. This, combined with the use of a uniform data structure for access to files and devices, known as a *channel*, and a uniform set of primitives results in a highly extensible operating system with a simple interface paradigm. The 9 I/O primitives are:

Attach - Connect a channel to the root of a file system and notify the file system

[6] Complex abstractions are represented as directories containing files representing different aspects of the abstraction. In the case of a process the files present include ones for memory, control and the text file

[7] An example of a software concept implemented as a file system would be Plan 9's representation of environment variables as files in the kernel resident file system

of which user is being attached.

Clone - Duplicate a channel creating a new channel that points to the same file or file system, as the original.

Walk - Perform a directory lookup on a file, and set the directory pointer to the next file or directory.

Stat - Get the file attributes of the current file.

Wstat - Alter the file attributes of the current file.

Open - Check permissions before opening file to allow I/O to be performed on the channel.

Read - Read from open file.

Write - Write to open file.

Close - Close open file.

As more than one file may be used to represent a device, it is possible to separate the control operation from the data transfer. By dividing device information this way, Plan 9, avoids the need for a call similar in nature to the ioctl found in UNIX.

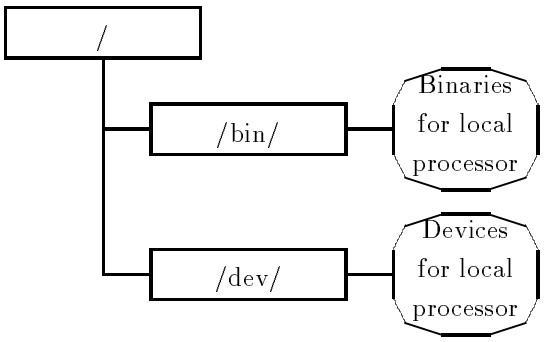


Figure 3.5: The Initial Name Space of a Process

Plan 9 creates a process group when a user logs into the system. This minimal process group (figure 3.5) contains a root directory, some binaries and some local devices. The system calls *mount* and *bind* are used to manipulate the name space.

The *mount* call adds new external file systems, and the *bind* call alters the arrangement of the name space. Directories in the Plan 9 system have the special property of being mounted *behind* another directory to yield the *union* of the two sets of files. Each directory in the union is searched, in turn, to find a file with a matching name. The first file found is returned. This feature replaces the UNIX *search path* concept in Plan 9.

The system is virtualized through control of the name space. The ability to alter any element of the name space, for a process, allows processes and objects to be moved from server to server within the system. Remote and local resources are easily interchanged by altering a process's name space. The virtual machine offered by Plan 9 is exploited when CPU servers are used as accelerators for processes. A daemon on the CPU server answers a request for a process to operate on the server by setting up a process group on the server. This allows resources, local to the CPU server, to be used by the process to be accelerated, without the process needing to be aware that it is running on a CPU server. Apart from an increase in speed, the process's environment appears identical to the process, regardless of whether it is operating on a CPU server or a terminal.

Local disk file systems have been avoided in the design of this operating system. The designers argue that local file systems require significant knowledge to administer on the part of the workstation user, which is frequently absent. In addition, as relatively few workstations export their file systems, the use of centralized file servers promotes file sharing and makes users independent of a specific terminal. The need for local systems is removed by using a combination of caching and high speed links, where necessary. By using high speed links between CPU servers and file servers, and large memory based caches on the file servers, a file access rate of a similar order of magnitude to the memory access rate has been achieved. Cache coherence is maintained through the use of a 64 bit file identifier. Half of this value is used to identify the file on a file server. The other half is a version number which is incremented each time the file is modified. The file identifier is returned each time a file is opened. If the version number component does not match, any currently cached pages are replaced, otherwise the cached pages are used. Where high speed

links are available, only pages of executable files are cached. On low speed links, a local disk is used as a large write through cache memory, for both executable and data pages. As this local disk acts only as a persistent cache of information held on the file server, it requires only limited maintenance. When the code or the user detects a problem with the disk, the disk is reformatted.

By using the file system paradigm to generate a uniform name space in which processes operate, Plan 9, provides a distributed system which uses a single conceptual model, allowing access to the full functionality of the system.

3.2.4 QNX

The QNX micro-kernel [Hil92, Var94] originated in 1982. It is currently in use in a large number of real time and distributed applications. The small code size of the kernel, and the ability to build systems without a file system, makes QNX well suited to embedded applications.

The micro-kernel supports 14 system calls. These form the interface to the functions of the QNX kernel: interprocess communication, process scheduling and interrupt dispatching. The kernel may optionally contain a network manager which supports low-level network communication. All system services are accessed through messages passed by the micro-kernel.

Message passing is implemented using a set of three blocking functions. The *Send()* call sends a message to a target process and blocks the current process until the target process executes a *Receive()* and a *Reply()* call for that message. If there are no pending messages, executing a *Receive()* call causes a process to block until a message is sent to it. The message is transmitted by copying between processes. No queuing is employed by these primitives. Message queues are supported through the use of *IPC servers* which are implemented using the three low level communication primitives. Processes can specify that messages be delivered in priority order rather than time order, and that the process executes at the priority of the highest-priority blocked process waiting for service. The use of these facilities prevents lower priority servers preempting a higher priority process by invoking the services of a process with even higher priority.

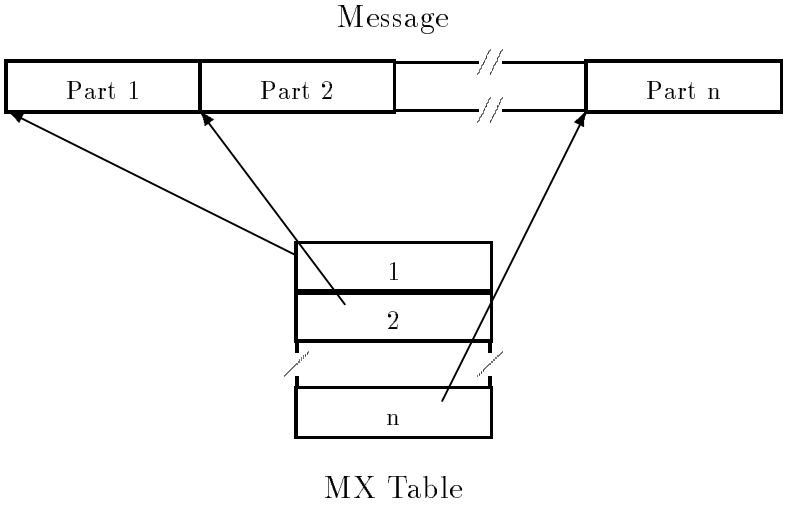


Figure 3.6: QNX Multi-part Messages

Multi-part messaging is supported by the low level communication primitives. An MX table (see figure 3.6) indicates where components of a message should be fetched from in the sending process's address space, or delivered to in the receiving process's address space. Direct support of multi-part messages reduces overheads encountered in other systems where messages must consist of contiguous memory locations. Systems which do not support multi-part messages typically use memory copies to gather elements in the transmitter, and scatter the elements in the receiver.

QNX supports the following scheduling policies[8]:

- Preemptive
- Prioritized context switching with round robin
- First in first out
- Adaptive scheduling

Sufficiently privileged user processes can connect an interrupt handler to an interrupt vector within the kernel. The handler, which runs within the process's address space, is called when an interrupt occurs. It has access to all the facilities of the user process. On completing its response to an interrupt, the handler has

[8]These policies are implemented in accordance to the draft standard: POSIX 1003.4 (real-time)

the choice of either returning to the kernel or waking the process which provides its address space. The ability to selectively start a process allows significant events to be noted immediately, and buffering to be used to improve efficiency. The micro-kernel conceals the presence of nested and shared interrupts, and any hardware dependent details from the user level interrupt handler. This feature provides a uniform interface thus simplifying the task of the programmer.

When present, the network manager is directly connected into the kernel providing efficient access to low level communications between QNX kernels. Transfers between local processes and remote processes are accomplished by issuing a message. The micro-kernel identifies the message and uses its private interface to the network manager to queue the message for transmission. After the dispatched message is received by the appropriate node, the network manager on that node passes the message to the local micro-kernel.

Multi-processor support is highly transparent as all services are available in response to messages, and message passing is handled by a network manager that is closely bound to the kernel. This simplifies the construction of distributed systems.

The only mandatory resource manager is the process manager - *Proc*. It supports process creation, accounting, environment inheritance, memory management and first level pathname management. Because a file system is optional within a QNX system, and as there are no other compulsory resource managers, *proc* initially owns the entire name space. *Proc* may delegate part of the name space to other managers. Name space delegation is conventionally implemented by having *proc* maintain a prefix tree of delegated pathnames, and ensuring that library routines using pathnames send a message to *proc*, which directs the routine to the manager corresponding to the entry with the longest matching prefix. File systems and network elements can easily be incorporated into this structure by delegating part of the name space to their managers. These managers are responsible for parsing the non-matching part of a pathname, and providing functions corresponding to requests directed to them by library routines.

The QNX system exhibits the flexibility typical of micro-kernel based operating systems. This flexibility is enhanced further by the use of a manager for network

interfaces and shifting name space management into a user level process. The use of message passing, as the basis of all system operations, provides inbuilt network transparency. This has resulted in a small operating system suited for use in distributed applications.

3.2.5 Angel

The Angel micro-kernel [MSWK93, MWO+93] employs a single 64-bit address space which is shared completely by the kernel and all processes executing on the system. The Angel kernel was influenced by perceived weaknesses and limitations in Meshix [OSW+92] - a conventional micro-kernel. The use of message passing within Meshix was identified as a major performance limitation. Lightweight RPC mechanisms were considered; however, the implementation of these mechanisms would reduce the strength of interprocess protection hence reducing the security of the system. Angel replaces RPC mechanisms with a shared address space model and implements secure LRPC mechanisms using shared memory.

The micro-kernel supports two major services: persistent virtual memory and virtual processor management.

The designers of Angel exploit the address space provided by a 64-bit processor to unify the functions of memory and the file system to produce a persistent store accessed through addresses. Memory objects are fixed in the address space. The designers term this a Single Address Space Architecture (SASA) as each process finds objects at the same place, within their address space. Improved data sharing is a major advantage of this style of address usage. Data always resides at the same address, allowing data structures to use addresses directly, and hence avoiding the need to encode data to remove pointers. The single unified interface reduces the complexity of programs. Only addresses are required to identify and use objects. The common interface is valuable on Distributed Shared Memory (DSM) machines as it eliminates the need for additional mechanisms to use resources located across the network. Shared memory allows LRPC mechanisms to be implemented through the use of shared objects which allows fast communication between co-operating processes at the security level provided to control access to memory.

The kernel supports the concept of *virtual processors*. A *virtual processor* behaves similarly to a UNIX process, except that whenever it performs an action that would block, it is 'upcalled'. The *virtual processor* can decide whether it can continue with some other task rather than surrender its time-slice to another *virtual processor*.

The kernel employs a control structure known as a *virtual processor* to represent the *virtual processor*. This data structure incorporates the state of a process and a list of *upcalls*.

An upcall is made by the kernel when an event which would block a *virtual processor* occurs. An upcall is implemented as a small data structure which conveys the type of the event causing the upcall and two further pieces of information specific to the upcall type. The upcall is delivered to the virtual processor if there is sufficient space in the upcall list, otherwise the upcall is ignored. The virtual processor structure specifies how an upcall is dealt with. The options for handling received upcalls are to discard the upcall, queue the upcall for later attention, or invoke a handler associated with the upcall type.

Threads are supported by user level code. The kernel has no explicit support for threads. The upcall mechanism is used to notify the user level thread package of events, including time based alarms, blocking events, and a change in state of locks.

Angel does not support the concept of user identifiers. Instead, it determines the right of a process to access an object based on what a process already has access to.

Access to objects within the Angel address space is controlled through the use of *Access Control Descriptors* (ACD). They describe the other objects which must be accessible to a process before this object may be used. A *biscuit* - a unique identifier for an ACD - is presented to the object manager to gain access to an object. The object manager confirms that the process has access to any objects listed in the ACD for the object the process wishes to load. If the requirements are met, the object manager allows access to the object.

Objects are composed of one or more pages of memory. Objects must be distinct, that is objects must not overlap, and an object may not enclose another object. Objects are created with a fixed length. It is necessary to copy the contents of an object to a larger object to effectively enlarge its size. Angel currently allocates

backing store to an object when a page is first modified. This allows sparse objects to be implemented with a minimum of backing store. However, it risks having a write to memory fail for a lack of backing store.

The Angel micro-kernel is a persistent system using a large flat address space to hold all objects. The fixed location of objects in the address space eliminates the need to make data structures independent of memory location, improving access speed and potentially simplifying the structures. The access mechanism of determining the right of a process to access an object, based on the ability to access other objects, allows the protection domain of a process - the set of objects it can access - to vary automatically.

3.2.6 Chorus

The CHORUS[9] distributed operating system [RAA+91, ARG89, Gie90] developed by Chorus systèmes uses a message passing based micro-kernel to support native programs and a UNIX based subsystem. Real time application support is incorporated into the nucleus.

Each *site* - a collection of one or more closely coupled processors within a CHORUS system - has a nucleus. The CHORUS nucleus (figure 3.7) consists of four major parts:

- The 'Supervisor' is machine dependent and is responsible for dealing with events generated by the hardware. The Supervisor incorporates the dispatch of interrupts, traps and exceptions.

- The 'Real-time Executive' allocates processors, provides fine-grained synchronization and preemptive priority-based scheduling.

- The 'Virtual Memory Manager' controls virtual memory hardware and local memory resources.

- The 'IPC Manager' provides asynchronous message exchange and RPC functions.

[9] CHORUS is a trademark of Chorus systèmes

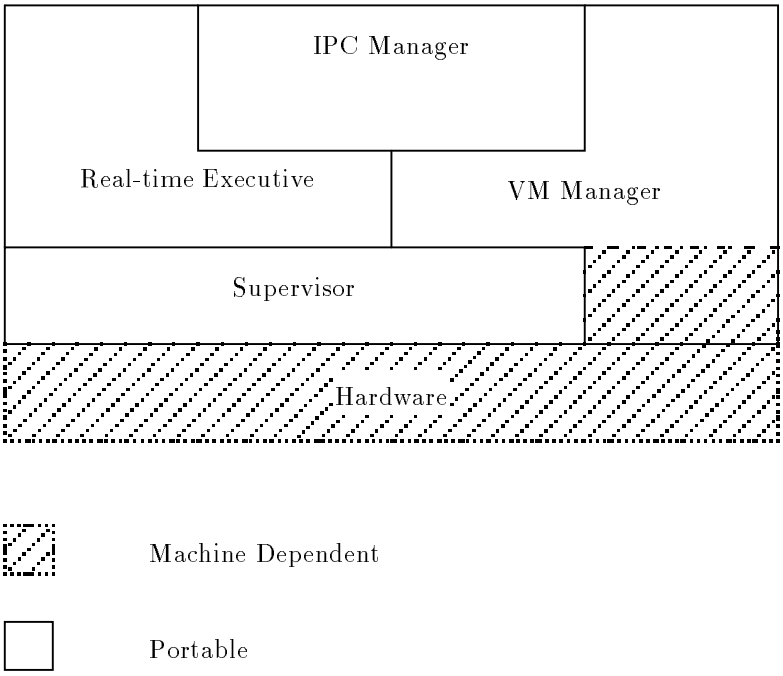


Figure 3.7: The CHORUS Nucleus

The four components of the nucleus are independent. The managers are distributed, with the users of the services being unaware of the separation of the components. Access to the functions provided by the managers, and between managers, is by the standard CHORUS IPC mechanism.

The nucleus uses a number of abstractions to represent objects it manages, and operations on those objects. Global naming is achieved through the *UI* - unique identifier - abstraction. The unit of resource allocation is the *actor*. The address space of an *actor* consists of a number of *regions*. The *thread* is the unit of sequential execution. Communication is conducted through *messages* directed to a *port* or *port group*.

A *subsystem* is composed of a number of servers operating cooperatively to provide a coherent operating system interface. Three abstractions are managed by both the nucleus and the subsystem servers. The *segment* is used to encapsulate data, a *capability* provides access control, and a *protection identifier* is used for authentication.

UIs are used to identify actors, ports, and port groups. These identities are

unique in both space and time, global, and independent of location. The CHORUS nucleus provides a service that allows the location of an entity with a UI.

A CHORUS actor encapsulates an address space divided into regions, a set of ports, and a set of threads. The regions are coupled with either local or remote segments. Threads are tied to a single actor, and share the resources of that actor with the other threads of the actor.

Actors are tied to a site, as are their threads. Only physical memory local to a site is used by an actor. The actors on a site have distinct user address spaces, and they share a single system address space. The shared address space is local to the site. Three types of actors exist:

User Actors are not trusted nor privileged

System Actors are trusted but not privileged. They may perform sensitive nucleus operations, but cannot execute privileged instructions.

Supervisor Actors are both trusted and privileged. They may execute privileged instructions as well as sensitive nucleus operations.

A port may be attached to only one actor at a time. However, ports can migrate from actor to actor allowing easy reconfiguration of services. Any thread within an actor may use a port held by that actor. Any thread that knows the name of a port may send a message to that port. Port groups are used to provide multi-cast and functional addressing. Ports may be added to, and removed from, port groups at any time. A number of addressing modes are provided to support port groups. It is possible to broadcast to all the ports in a group, to send to any one port in a group, to send to any one port on a given site, and to send to any one port on the same site as the sender.

A message consists of a contiguous string of bytes copied from the sender's address space to the receiver's address space. The copy is optimised. If possible, a page descriptor is moved between the sender and the receiver; failing that a copy-on-write technique is employed.

CHORUS supports both asynchronous IPC and synchronous RPC mechanisms.

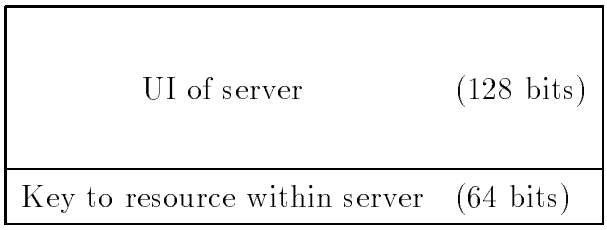


Figure 3.8: A CHORUS Capability

A capability (see figure 3.8) comprises the UI of the server which manages the object, and a key which uniquely identifies the object. Capabilities are used as global names for objects which are not directly implemented by the CHORUS nucleus.

Each message is stamped with the protection identifier of the source actor-port combination. This allows the receiver of the message to verify the identity of the sender.

Deferred copy techniques and local caching are employed by the CHORUS microkernel to improve performance. These techniques have a significant impact on operations which copy large amounts of data between actors, and on IPC and IO operations which move small amounts of data between segments.

The model of distributed computing employed by CHORUS ties processes to clusters of tightly coupled processors and employs a position independent addressing mechanism to allow the delivery sites of messages to migrate. This mechanism avoids the difficulties of shifting processes while it provides a way of distributing the work load and allowing services to migrate. The performance of message passing is improved through the use of caching and deferred copying.

3.3 Capability Based Operating Systems

A capability based operating system uses capabilities to control access to objects and services. The capability is used as the primary mechanism for referring to an object. Possession of a capability implies the right to access an object or service.

Five capability based operating systems are covered here: Monads, KeyKOS[10] Grasshopper, Opal and Mungi. In each of these systems, capabilities are the fun-

[10]KeyKOS is a trademark of Key Logic, Inc.

damental access control mechanism. However, these systems display widely varying properties, and exploit capabilities in differing ways to provide the facilities of the operating system.

3.3.1 Monads

The Monads project [RA85a, Geh82, Kee82] ran from 1976 to 1985 at Monash University. This project attempted to provide an architecture suitable for the development of large and complex software systems. Special purpose hardware was designed and built to support the Monads operating system.

The designers of the Monads systems applied the principle of *information hiding* to the decomposition of complex systems into modules. The modules communicate only through the defined module interface, and the internal data structures of a module are inaccessible to other modules. The module interface consists of a set of *exported* functions which may be called from other modules.

Under Monads, capabilities are viewed as protected pointers - a pointer which cannot be modified by the process - to objects which exist in a large uniform address space. Two classes of objects are supported: modules and segments. The segment is the base unit of storage, and cannot be shared between modules. A module is a collection of segments which store all the module's information. Module capabilities are used to refer to other modules. Segment capabilities are used to address within a module.

Every memory access within the Monads architecture makes either explicit or implicit use of a capability. Thus all memory accesses are checked at the instruction level to ensure that appropriate rights are held to perform the operation.

A uniform paged virtual memory is used to hold all the data contained in a Monads system. The virtual memory is accessed through the use of large addresses which identify single bytes[11]. Segments are defined by a start address and a limit. They need not be page aligned. (see figure 3.9)

Each segment is intended to contain a single logical entity[12]. Segments have ex-

[11]Monads-PC supported 60 bit addresses [RA85b]

[12]An array and a procedure are examples of logical entities

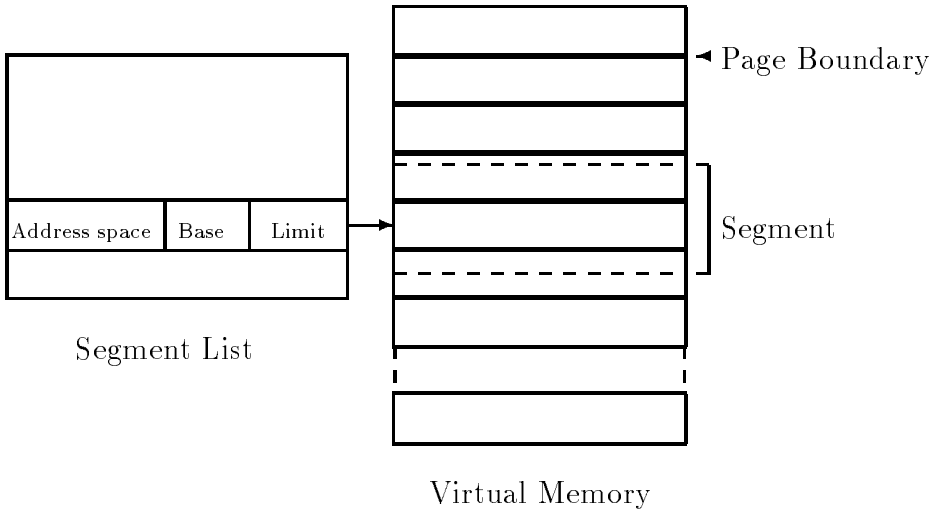


Figure 3.9: Mapping Monads Segments to Pages

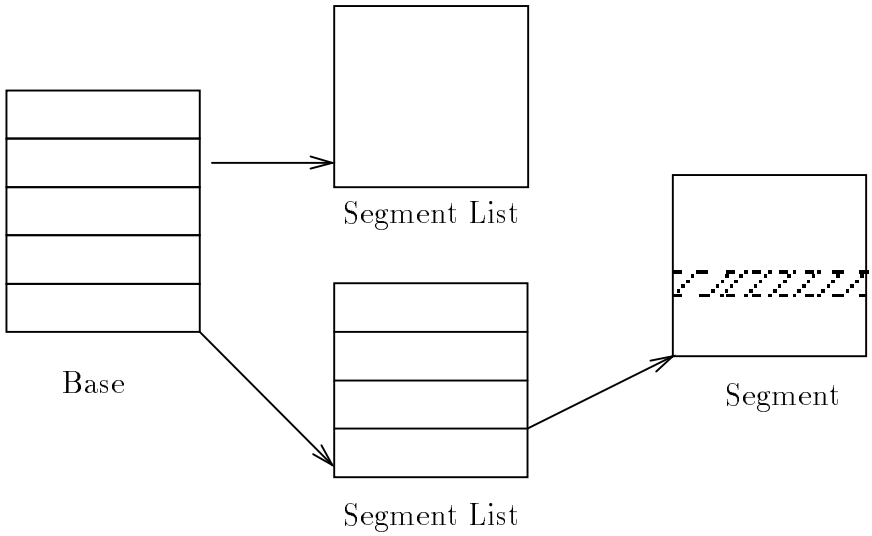
clusive use of the virtual address space they occupy. They are created in a previously unused portion of the address space, and when they are destroyed, that portion of the address space becomes unusable.

Each process is associated with a number of *base registers* which point to lists of capabilities know as *segment lists* (see figure 3.10). Segments in these lists are available for use by the process. A process cannot directly modify either a base register or a segment list. Memory can only be accessed by specifying a base register, segment number and offset. A limited number of *capability registers* are provided to increase efficiency. This allows memory addresses to be specified by a capability register and an offset.

The inter-module call mechanism performs several tasks. The call instruction:

- loads the base registers with values appropriate for the called module
- creates a local data segment list and associated segments on the stack
- loads a base register with the segment list of passed parameters
- transfers control to the called module

This limits a module to its own data and any data passed to it. The return call



<base, segment number, offset>

Figure 3.10: Addressing Memory Under Monads

removes local segments and restores the previous module's environment before returning control to the calling module.

Four types of program data are identified in the Monads system: code-related data, local data, permanent data, and retained data. Code related data is created at compile time, and typically consists of constants embedded in the code. Local data is used for temporary storage within a procedure. It is created on entry to a procedure, and destroyed on exit from a procedure. Permanent data[13] is data associated with a module. Several different versions of this data may exist, each being associated with a different instance of the module type. The permanent data persists until the module is destroyed. Retained data is associated with a particular invocation of an instance of a module. The retained data persists between calls to a module, and is destroyed when a program terminates.

Each module is required to have two standard procedures: *CREATE* and *OPEN*. These procedures are used to support permanent data and retained data. The

[13]Permanent data would be stored in files in a conventional operating system.

CREATE call is used to generate a new instance of a module. Creating a module is performed by making a data segment list and associated data segments. This allocates space for a *permanent data* entity which can be shared by a number of processes. The *OPEN* call is made on the first invocation of a module by a program. The call consists of two components: the first creates the retained data segment list and segments, and the second component, removes the retained data segments. The second component of the open call is made when the program terminates.

Monads employs an in-process architecture. The operating system does not run as a separate process to user processes; instead operating system functions are used by calling privileged procedures. Monads supports only user level processes; all operating system functions are accomplished through calls on privileged procedures. There are no auxiliary operating-system processes.

Privileged procedures are protected by the same means as normal procedures. The data of a procedure is inaccessible while the procedure is not running. While a procedure is executing, its data is available to the code of the procedure, but the data must cease to be available when the procedure is no longer executing. The data of a procedure includes information held on the stack. The Monads hardware supports this mode of operation. This level of protection is sufficient to protect operating system functions.

The Monads project produced a number of systems, all using specially constructed or modified hardware to implement an environment well suited to supporting small modules, with strict isolation between the modules, except through the parameters of a set of exported functions. Capabilities provide both unique names and a mechanism to implement access restrictions. A C-list based architecture is used to protect capabilities from user process manipulation.

3.3.2 KeyKOS

Development of the KeyKOS system [BFF+92, Har85] began in 1975 and has been in use in production systems since 1983. KeyKOS is a capability based operating

system, with check-pointing built into the kernel. The micro-kernel[14] was developed to support a secure environment, with data sharing, high reliability, and accurate pricing. An abstract machine interface is presented to each application program. The abstract machine interface may be exploited to implement either a KeyKOS service or an operating system emulation. EDX, RPS, VM/370, a subset of MVS and UNIX have been implemented.

The original application of KeyKOS was to support British Telecom's Tymnet service. Isolating mutually antagonistic users, supporting highly accurate accounting, and providing 24-hour uninterrupted service were the three major priority requirements placed on the operating system.

The architecture of the micro-kernel is based on 3 concepts: a stateless kernel, a single-level store, and capabilities.

All the micro-kernel's state is derived from information held in persistent state. The information cached in the micro-kernel may be in a different format to the persistent information to increase efficiency. The private information of the micro-kernel may be discarded at any time, as it can be reconstructed as necessary from information held in the persistent data elements, known as nodes and pages. The elimination of critical state in the kernel eliminates the need to checkpoint the kernel, and avoids the need for dynamic allocation of kernel storage.

All the data of a KeyKOS system is stored in a persistent virtual memory system. Only the micro-kernel is aware of which pages are present in main memory. The paging system is tied to the administration of checkpoints, and periodic system-wide checkpoints are used to guarantee the persistence of data. Both processes and data are persistent. Service interruptions appear, to application processes, only as unexplained jumps in the real time clock.

Capabilities[15] are central to KeyKOS's operation in that they are used to control access to objects and the sending of messages. No other mechanisms are present to complicate the implementation. Objects in the system are exclusively referred to by their capabilities, and possession of a capability implies the right to use the capability

[14]The designers of KeyKOS identify their kernel as a 'nano-kernel architecture', but offer no explanation as to how it differs from a conventional micro-kernel architecture

[15]The authors of KeyKOS uses the term 'key' instead of capability for brevity

and to pass the capability to a third party. The right to create a capability is privileged; however, the right to duplicate a capability is available to all applications. To prevent forgery, capabilities are segregated so that only the micro-kernel has access to the capability.

Encapsulation of objects is enforced by the kernel through the use of message passing and capabilities. Mutually suspicious users are protected from each other by the capability mechanism. For a user to gain access to a service or object, it is necessary for a holder of a service or object to disclose the required capability.

Multiple capabilities may refer to a single process. An 8-bit field in a capability is used to identify the class of capability used to send a message to the process. When a process hands out a capability, it can set this field to a known value. This field allows the process to partition clients of the process into either service classes, or privilege levels.

Six fundamental objects are supported by the KeyKOS kernel:

Devices: Device drivers are typically split into two components. Low level hardware drivers are implemented in privileged code and perform the tasks of message encapsulation and hardware register manipulation. The high level driver is typically implemented as a KeyKOS process[16].

Pages: The simplest object in a KeyKOS system is a page. The size of a page is machine dependent[17]. At initialization the number of pages that a system can manage is fixed.

A page responds to *read* and *write* messages. If a page is mapped into a processes address space, then loads and stores on locations within a page are equivalent to the operation of *read* and *write* messages. If a page is not present in memory when a message is sent to it, then it is brought into memory before performing the operation on the page.

Each page is known by at least one *page key*, and the page has at least one persistent location on disk known as its *home location*.

[16] Except where performance would be inadequate

[17] In all current implementations of KeyKOS pages are 4 kilobytes in size

Nodes: KeyKOS segregates capabilities from direct scrutiny by user processes. It stores capabilities in nodes[18]. A *node key* is a capability that gives access to a node. It is used to add and remove capabilities from a node.

Segments: Address spaces are defined through the use of segments which represents a collection of pages or other segments. Segments are sparse and are not required to be contiguous. They are implemented as a tree of nodes, with pages as the leaves of the tree.

Meters: A *meter key* entitles the holder to the amount of CPU time held in the *meter* corresponding to the capability. There is no guarantee that the CPU time will be allocated in a contiguous unit.

Domains: A domain consists of 16 general key slots, a number of special key slots and all the non-privileged state of the hardware available to a KeyKOS process. When a slot in a domain is loaded with a capability, the process executing within the domain is deemed to *hold* the key. The special slots for a domain include: an address slot which holds the capability for the segment acting as the address space for the domain, and a slot for a *meter key* which provides execution time for process executing within the domain.

Under KeyKOS a message consists of a parameter word, a string of up to 4096 bytes, and four capabilities. Only capabilities held by the sender can be sent. There are 3 mechanisms available for sending messages: *call*, *fork* and *return*. The *fork* mechanism sends a message to the recipient, and does not wait for a reply. The *call* mechanism generates a *resume key* for the sender, and dispatches the message to the recipient. The sender is suspended, and refuses messages, until it receives a message sent using the *resume key*. The *return* mechanism sends a message, leaving the sender able to receive messages.

Messages are not buffered. If the recipient of a message is unable to handle a message immediately, then the sender of the message is deferred until the receiver is ready.

[18]In all current implementations of KeyKOS nodes have sixteen slots

On receipt of a message, the receiver of the message may choose which parts (parameter word, string, or keys) of the message it accepts, discarding the remainder of the message.

Periodically a KeyKOS system checkpoints. At this stage, all processes and I/O activities are stopped, any dirty pages are transferred to the current checkpoint area on disk, the alternate checkpoint area is made current and the processes are permitted to resume. Pages are then migrated from the first checkpoint area back to their home locations. By ensuring that a second checkpoint does not occur until the first checkpoint is handled, the system remains in a non-corrupt state.

Exceptions are managed through the use of *keepers* which are associated with domains, segments and meters. A message is sent to the appropriate *domain keeper* if an exception occurs. The *keeper* may either terminate the program, supply an answer and allow execution to continue, or restart the instruction. *Segment keepers* are invoked if an invalid operation or a protection violation is performed on a segment. A *meter keeper* is invoked when the associated meter runs out. This can be used to implement complex thread and process scheduling mechanisms.

The use of capabilities for access protection and check-pointing for ensuring system consistency allows KeyKOS to provide an environment where applications are secure from both external failures and internal attacks.

3.3.3 Grasshopper

The Grasshopper operating system [DdBF+94b, DdBF+94a, LDdB+94, DLR95] is an experimental operating system that seeks to provide a scalable and efficient persistent environment using conventional workstation hardware. The operating system is being developed by researchers in the computer science departments at the University of Sydney and the University of Adelaide.

Unlike Monads (see section 3.3.1), Grasshopper, is designed to run on conventional hardware, using the existing page translation hardware to support access control and memory mapping. A direct consequence is that access control and memory structuring occurs in multiples of pages.

The designers of Grasshopper identify two principles which define 'orthogonal

persistence'. The requirements are that objects exist for the same period as the object is required and that objects are manipulated in the same manner regardless of their longevity. To provide an environment which supports orthogonal persistence three fundamental abstractions are employed by the operating system: *Containers* are the abstract representation of storage, *loci* represent actions within the system, and *capabilities* are used to represent a right of access to an object.

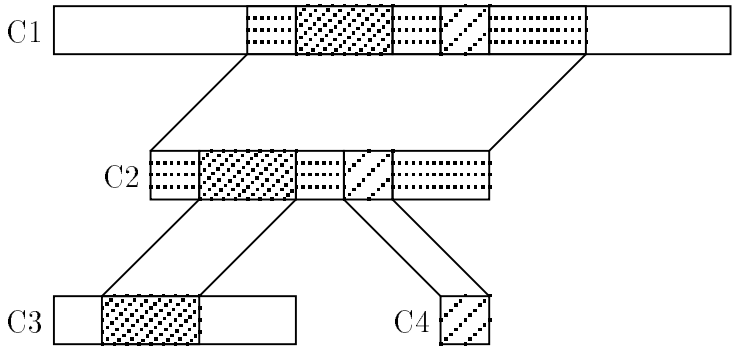


Figure 3.11: Mapping containers under Grasshopper

Containers provide all access to storage within the system. They are persistent and may be of arbitrary size. The contents of containers are derived from 2 sources: sections of other containers and information supplied by a *manager*. Containers may map in segments of other containers (see figure 3.11) to construct a directed acyclic graph of dependencies on other containers. The elimination of cycles ensures that there exists a container that is responsible for the provision of data. When a reference is made to a location within a container (resulting in a page fault), the kernel determines which container is responsible for the delivery of the information and calls the appropriate manager to make the information available.

Managers provide data when it is not resident in memory. They are normal user level programs which reside and execute within their own containers. Managers provide pages of data which are stored in a container, respond to access faults and handle data removed from memory by the kernel. It is the responsibility of the manager to maintain the coherence and integrity of the data of the containers that they manage.

'Manipulative managers' store data in a different form on permanent store to

the form that is present to a locus in a container. Three examples of *manipulative managers* are:

Swizzling managers may be used to support a container that is larger than the address space of the host hardware. When address fault occurs in a container managed by a swizzling manager, a page of data is provided which contains a set of addresses. When one of these addresses is dereferenced, the swizzling manager maps in the correct data.

Encrypting managers encrypt the data placed on the permanent store.

Compressing managers compress the data placed on the permanent store resulting in storage savings.

The presence of manipulative managers clearly distinguishes the container concept from the concept of an address space.

Loci are the active elements which manipulate the contents of containers. In principle, a locus always executes within a single container, known as its *host container*. The virtual addresses generated by the locus are used to access the contents of the host container. By using the *mapping* mechanism, the locus can make use of the contents of other containers. Any number of loci are permitted to execute within a container, allowing the operating system to support multi-threaded programs. Loci are maintained as persistent entities by the Grasshopper kernel.

A locus can change its host container by invoking another container (see figure 3.12). The locus enters the container at a location known as the *invocation point* which is specified as an attribute of the container. The single entry point forces the locus to execute code that is under the control of the invoked container. This allows the invoked container to ensure its own security. A parameter block is available which allows a small amount of information to be carried between the containers by the locus. Invocation is a low cost operation as the minimal parameter block is the only context transferred to the invoked container. Larger quantities of information may be conveyed through the use of an intermediate container.

Invocation is analogous to procedure calls in that loci may make invoke other containers, and issue a return which takes the locus back to the invoking container.

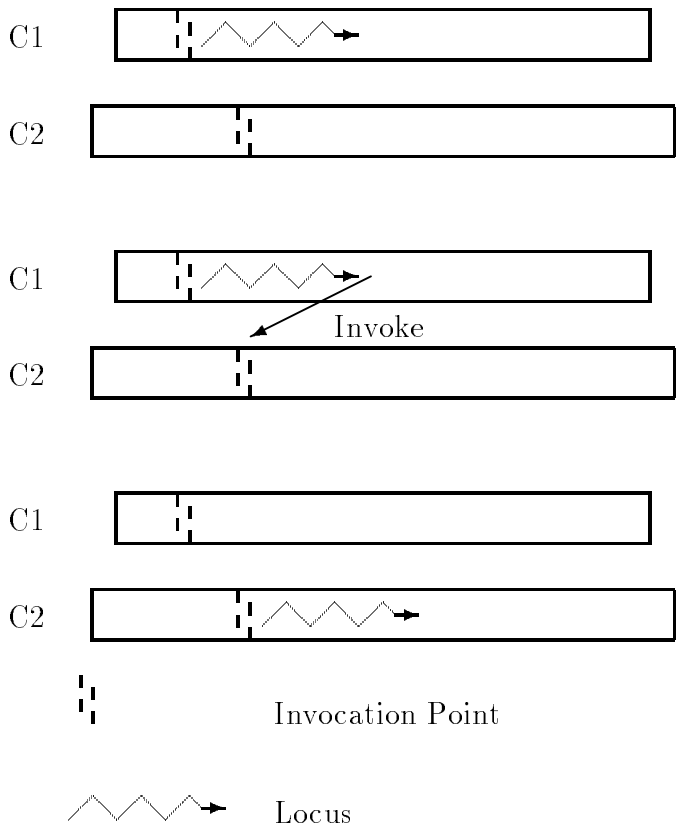


Figure 3.12: Invocation under Grasshopper

The kernel maintains a call chain of invocations. As some loci may not require to return to the container from which they were invoked, a mechanism exists to inform the kernel that no return chain need be kept.

Locus private mappings allow loci to share their host containers while retaining private data which is visible only to a single locus. These mappings take precedence over host container mappings. Locus private mappings simplify the implementation of multi-threaded programs, and provide a mechanism for a locus to keep information secret from other threads executing in the same container. This feature is typically used to implement stacks, without the need to ensure that stacks are separated throughout the address space perceived by the loci in a container.

Grasshopper employs capabilities to provide unique naming and access control. Capabilities are stored in list structures segregated from containers and loci. Segregating the capabilities simplifies the task of garbage collection. A reference count is maintained on each object and when all capabilities relating to the object are deleted, the object is removed. Segregation of the capabilities allows the kernel to keep an accurate count of valid capabilities for an object.

Conventional hardware is used to support Grasshopper's protection system. The use of capabilities to refer to coarse grained objects, specifically containers and loci, allows conventional hardware to provide adequate security without an excessive overhead.

A locus has access to a number of sets of capabilities.

1. The set of capabilities contained in the locus' private list of capabilities.

2. The set of capabilities contained in the list of capabilities associated with the host container.

A locus can move capabilities into and out of the lists, or perform an operation using a capability through functions provided by the operating system. Specific capabilities are identified by nominating a capability list and a key.

Managers are capable of, and responsible for, the maintenance of a consistent recoverable state of their containers.

Grasshopper provides an environment where processes and objects are persistent. The abstractions provided by the Grasshopper system decouple the address

space from the active agents within the system, and provides a mechanism for transparently transforming the contents of an address space based on the actions of an agent.

3.3.4 Opal

Opal [CLFL94] is a capability-based single address space operating system (SASOS). It is under development in the Department of Computer Science and Engineering, University of Washington, Seattle. Like Angel (see section 3.2.5), this operating system exploits the appearance of 64-bit address space architectures to provide a single address space in which objects are uniquely identified by their addresses. Password-Capabilities are used to protect access to objects. Opal is currently implemented as a prototype based on the Mach 3.0 micro-kernel.

The Opal system is based upon the principle that addresses have a unique interpretation for all applications. That interpretation is independent of the user of the address; hence a thread may name any data item. Protection - the control of access to an object - is managed through *protection domains* which permit a thread to have access to a specific set of pages at a given instant.

Single address space operating systems have the advantage of ease of sharing data-structures containing pointers between threads (processes). For private address space operating systems to share structures containing pointers it is necessary to convert the data to an intermediate representation which is shared, or to coerce the processes using the shared data-structure to load the data at a fixed known location in their address space. Poor compromises - typical of conventional systems - between protection, performance and integration, are avoided by SASOS. These benefits follow from the twin factors: eliminating the need to transform data being shared between threads; and employing protection based on protection domains which give specific permission to a thread to access shared data.

Under Opal, a *segment* is defined as a continuous extent of virtual pages. The virtual address of the segment is fixed at the time of allocation and remains unchanged during its existence. Segments are the base unit of storage and protection. Non-transitory data is held in segments marked *persistent*.

A *protection domain* consists of a set of access rights to segments at a given instant in time. A *thread* executes in a single protection domain and has the access rights conferred by that domain. Threads may execute simultaneously in a given protection domain, and hence have access to the same collection of segments.

Password-capabilities are used to control access to resources such as protection domains and segments. Opal capabilities are 256-bits in size and confer permission to operate on an object in specific ways.

Given a capability, a thread can *attach* that segment to its protection domain making the segment represented by the capability available to all the threads of the protection domain. A segment can be made inaccessible by *detach*ing it.

Calls are made across domains through *portals*. A *portal* consists of a fixed entry point into a domain identified by a unique 64-bit *portalID*. Any thread knowing a portalID can transfer control into the domain associated with the portal.

Opal implements password-capabilities as an extension of the inter-domain communication mechanism. A capability consists of a 64-bit portalID, a 64 bit object address and a 128-bit random check field. The portalID is used to make a call on the server which manages the segment or domain represented by the capability. The check field performs the functions of identifying the set of rights conferred by the capability, preventing forgery, and permitting revocation. The server either grants or denies access to the resource represented by the capability.

In a password-capability system it is not possible to distinguish an instance of a capability from ordinary data. Thus these systems cannot determine when there are no references to an object, and hence determine a time when it is safe to destroy the object and reclaim resources devoted to it. Opal uses a reference count associated with each object to determine when it is safe to destroy an object. The reference count is incremented each time a segment is attached, and decremented when a segment is detached. The reference count keeps track of the number of protection domains which can make direct use of a segment[19] rather than the number of capabilities existent for a segment. In addition, a segment can be made *persistent* causing it to remain after the last detach.

[19] An alternative interpretation is that the reference count "... indicates the number of entities that have registered an interest in a resource..." [CLFL94]

The basic reclamation mechanism is subject to errors. It is possible to arrange both the premature release of resources and the permanent commitment of resources. Opal implements *reference objects* and *resource groups* to overcome these errors. *Reference objects* allow private reference counts to be created for each group of entities who wish to access a shared object. These groups are typically composed of mutually trusting threads. The groups are suspicious of other groups which have access to the shared object. Only when all the reference counts are exhausted will the object be deleted. The reference counts ensure that the object persists until all suspicious groups have relinquished an interest in the object. *Resource groups* are used to release resources and reference counts held by an entity when the entity ceases to exist. The capability for a resource group is passed as a hidden argument in any call that creates a resource or increments a reference count. When a resource group is destroyed, the references are released. A thread may alter its resource group, at any time. This mechanism ensures that all resources can be reclaimed even if the normal reclamation mechanism fails.

The Opal system provides a single address space in which threads are able to efficiently share structures containing pointers. Access control is performed using password-capabilities supplemented with resource groups - to assist in garbage collection - and reference objects - to assist in sharing between mutually distrustful threads.

3.3.5 Mungi

Mungi [HERV94] is a SASOS under development in the School of Computer Science and Engineering at the University of New South Wales. The goals of their system are similar to Opal (see section 3.3.4) in that both support direct sharing of objects containing pointers through the use of a 64-bit address persistent address space.

Mungi is designed to support a medium sized network of homogeneous machines. It provides a single virtual address space transparently distributed over the nodes. Password-capabilities are used for naming and access control of objects. Objects have a paged sized granularity. No special hardware is required by the system to support the operating system as the existing page translation hardware provided by

the processor is used.

Under Mungi, objects are defined as contiguous sequences of pages. Objects are the base unit of allocation and protection. The operating system locates objects via the *Object Table* (OT). The OT's entries list an object's address, size, accounting information, and passwords with corresponding access rights. The OT is distributed and is organised as a B^+-tree. The OT is constructed from a number of ordinary objects. These objects are distributed across nodes by partitioning the address space according to node and placing the local nodes OT bucket in the address range associated with the local node. By partially replicating the nodes of the tree with read-only copies, the number of accesses required between nodes can be reduced.

Pages may exist in one of six states on a given node: resident, on-disk, remote, zero-on-use, unallocated, and unknown. The node holding the master copy of an allocated page is defined as the *owner* of the page. Additional read-only copies of the page may exist. Pages can migrate and hence their owner may change. A *location hint* is stored for non-local pages. If no *location hint* is available, then the kernel may infer one based on the *location hints* for surrounding pages. When referring to a non-local page, the node contained in the *location hint* is contacted. If the page is not at that location, either the node uses its location hint to forward the message, or a broadcast message is sent to locate the node.

Attempting to perform a read operation on a page, which is not present on the node, results in a read-only copy of the page being transferred to the node. Attempting to perform a write operation on a page, which is not present on the node or present as a read only page, results in the page's contents and ownership being transferred to the requesting node.

The global address space is partitioned. A partition is mounted on a node which is made responsible for the creation and deletion of objects which appear within that partition. This mechanism is employed to simplify the management of memory by ensuring that knowledge of unallocated pages is only required on the creating node. The creating node has no information about the location of pages from the segments it manages apart from that gained through the reference mechanism available to all nodes.

To improve efficiency, Mungi supports 3 classes of data objects:

- Transient and unshared
- Transient and shared
- Persistent

Each data class is allocated in a separate partition of the address space. By ensuring that transient objects are allocated in a section of the OT that is local to the node and will remain so, the speed of creation and destruction of objects is enhanced.

Capabilities under Mungi consist of a 64-bit address and a 64-bit password. Associated with each capability is an object and a set of rights permitted over the object, drawn from the collection: read, write, execute and destroy. Capabilities can be derived and revoked.

Every kernel in the system has access to an object which contains the *capability tree* (Ctree - see figure 3.13). A Ctree is constructed from *protection nodes* (Pnode). A Pnode may contain a pointer to a list of capabilities (Clist) and/or a pointer to a *protection fault handler*. Each user is assigned a pointer to a Pnode which is used to define the process's *regular protection domain* (RPD). This pointer points to a leaf Pnode. All capabilities found in Clists on the direct path between the leaf node and the root are accessible to the user. Protection fault handlers can be supplied by the user to supply an alternative search strategy. These handlers either return a capability or a failure indication. If a capability is returned then the kernel makes use of that capability otherwise the search towards the root of the tree is continued.

Protection is managed through the use of *Active Protection Domains* (APD). Each APD consists of a list of Clists and protection fault handlers. It is constructed when a user logs in from their RPD. User processes can modify APDs by adding or removing Clists or handlers. Processes can be created which operate in a limited protection domain by *locking* the APD in which the process operates. This allows untrusted code to be used in a controlled environment.

Mungi also provides a facility to allow code to temporarily change protection domains. This feature is analogous to the UNIX setuid facility. *Protection Domain Extension* (PDX) occurs when procedure in a PDX object is called. The kernel

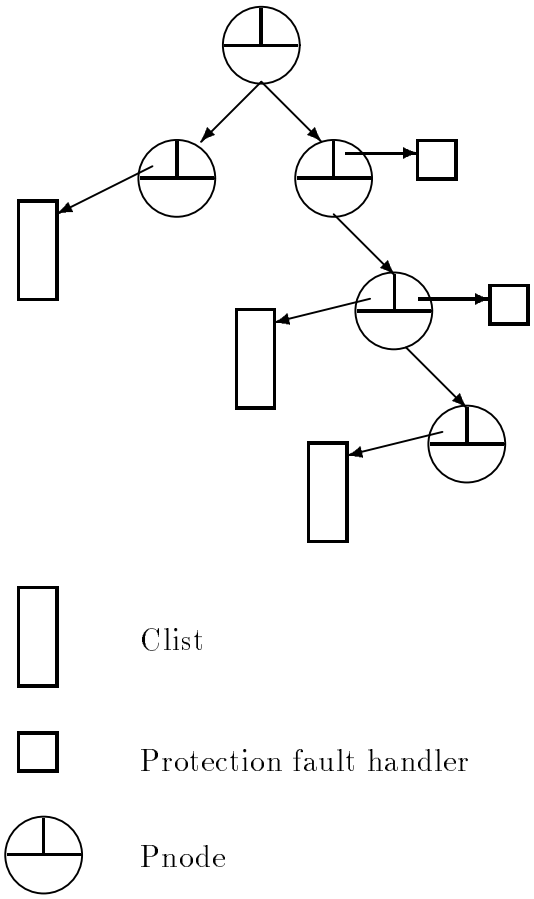


Figure 3.13: Capability Trees Under Mungi

verifies that the address of the procedure matches a valid entry point and proceeds to push a Clist pointer associated with the PDX object onto the APD. This gives the PDX procedure access to a set of capabilities unavailable to the procedures in the original APD. On return from the PDX procedure the Clist pointer is popped off the APD.

Mungi provides a sophisticated capability based protection system on a SASOS. By combining the features of restricting an APD before invoking a procedure and using PDX procedures, it is possible to construct environments with protection domains tailored to minimise access between mutually non-trusting processes.

3.4 Observations and Trends

Operating system design is influenced by a large number of often conflicting factors. Most prominent among these factors are the choice of abstractions made by the system designer, the functionality of the target hardware, and user expectations. Examination of these factors provides insight into trends in the design of operating systems.

The following trends are identified:

- The trend towards supporting large numbers of platforms and the impact of this design decision on both hardware and software design
- The emergence of small kernels and micro-kernels as the preferred design path over monolithic kernels
- The use of capabilities to support other paradigms
- An increase in support for distributed computing environments
- An increase in research into persistent systems contrasted with the low rate of acceptance of persistent environments in commercial computing
- The provision of memory objects as directly manipulable and shareable objects between processes.

Operating System	Addrs size[1]	Kernel Type[2]	Paradigm[3]	Platform[4]
UNIX	16	M	F	680x0, 80386, Alpha, Rx000, Sparc, VAX, etc†
Amoeba	32	u	cDOT	68020, 80386, MicroVax II, Sparc
Mach	32	u	cDOT	80386, IBM 370, IBM RT-PC, NeXT, Sun 3, VAX, etc†
Plan 9	32	S	FD	SGI Power Series, 68040 Nextstation, MIPS Magnum, Sun SLC, AT&T Safari PC
QNX	32	u	FD	80486
Angel	64	u	SDPOt	SunOS‡, Tadpole M88K
CHORUS	32	u	cDOt	680x0, COMPAQ 386/486, R3000, Sparc, Inmos T425/T805
Monads	60	M	CPO	Custom Hardware
KeyKOS	32	u	CPO	680x0, 88x00, IBM 370
Grasshopper	32	*	CPOT	Sun 3, Alpha
Opal	64	*	CSOT	R3000×, Alpha×
Mungi	64	*	CSDPOT	*

[1] Minimum address size required in native implementations

[2] Kernel Type: u - micro-kernel, S - small kernel, M - monolithic

[3] Paradigm: S - single address space, C - capability based, c - uses capabilities,
F - access control via file system, D - distributed, P - persistent, O - Object based
T - supports threads with concurrent execution on multiprocessor,
t - supports threads with single thread per process active

[4] Partially supported processor families have the least capable suitable member listed

† Many other systems

‡ Hardware is emulated on SunOS based server

× Hosted by a Mach server

* Information not available

Table 3.1: Summary Table for Reviewed Operating Systems

- The emergence of operating systems supporting the thread paradigm.

The majority of the operating systems reviewed in this chapter are supported on a large number of platforms (see table 3.1). Support of a wide range of platforms requires the operating system to conceal the differences between platforms by presenting the application developers with a *virtual machine* independent of the implementation platform. The trend of providing uniform environments over dissimilar hardware is not recent, although it has significant implications for both hardware platforms and operating system software. The prime consequence is a tendency for operating system designers to use only features found on the least capable of the target processors. As the system needs to run acceptably on all target processors there is an additional requirement on the designer to make the system run well in the absence of any processor specific features which might improve performance. This has led to a self reinforcing trend in both hardware and software design where the software designer expects only a minimal set of features provided by the hardware and the hardware designer provides only a minimal set of features. Hardware which supports segments evidences this trend. Only 3 of the commercial processor types listed support segments: 80386, IBM 370, and IBM RT-PC[20]. The remainder of the architectures support only paging. A number of the operating systems in this survey may benefit from the fine grain access control provided by segments, however, the lack of wide spread architectural support has led designers to use coarser grained paging based schemes to provide access control and to typically ignore segmentation hardware even when present.

New operating systems tend to be Micro-kernels or small kernels. There are several reasons for this trend:

Verification - users are now requiring verification of software in *mission critical* applications. Monolithic kernels are not well suited to verification as they tend to be large and complex. In addition operating system components tend to have multiple points of interface. This increases the complexity of the analysis required to *prove* the system. Small kernels offer a limited number of services and tend to be less complex. This reduces the amount of critical

[20]Forerunner of the Power-PC architecture

code which needs to be verified at the core of the system, simplifying the task of proving that the basic functions of the operating system are correct. By delegating functions found in monolithic kernels to user level servers (for example: management of the file-system) which have well defined interfaces, the task of verifying these functions is simplified.

Cost - By reducing the size of the kernel code the task of writing and maintaining the code is reduced.

Flexibility - As only essential services are provided by system level code it is possible for services to be written which suit a given work environment without encountering any loss in performance compared to a standard system service.

Multiple Personalities - Micro-kernels allow the use of multiple *personalities* of operating systems on a single system. As the operating environment perceived by the user is provided by user level code it is possible to run several different environments on a single system.

The trend towards developing small kernels is likely to continue as it reduces development costs and increases flexibility. The trend is not without a price as there is a potential loss of efficiency when servers communicate. Research is being conducted into reducing the cost of communication in micro-kernel systems, however, it is clear that if similar techniques are applied to monolithic kernels, any speed advantage gained by micro-kernels would be negated. This has led to the development of a class of operating systems which employ a *micro-kernel architecture*. These operating systems are essentially monolithic kernels with a modular design discipline. In principle, this simplifies the task of writing and verification to the same level as a micro-kernel. In practice, unless a strongly typed language with rigorously checked pointers is used, the absence of address space separation between modules of the operating system allows unforeseen interactions to occur, negating some of the advantage over a standard monolithic kernel.

Table 3.1 indicates that the capability paradigm has been taken up by a number of current operating systems. The use of capabilities is likely to continue as they

provide a mechanism which solves a number of problems. Capabilities offer the following advantages:

- storage allocation and protocols are simplified as capabilities are of fixed size.
- verification of the sender of a capability is typically unnecessary which simplifies the implementation of system security.
- unique naming scheme simplifies the generation of names.

Operating systems supporting distributed computing environments are gaining importance. This points to a perceived need to provide facilities which allow processes to co-operate over multiple machines. The tendency towards multiple processors is cost driven, as it is typically less expensive to purchase a number of less capable machines rather than a single highly capable machine.

Persistence is a current research topic. The KeyKOS operating system is the only commercial persistent system reviewed here. Other commercial persistent operating systems exist, such as IBM's AS/400 [ST89]. However, there is relatively little documentation relating to their hardware and operating systems publicly available. It is also possible to build persistent systems over a Mach micro-kernel. Persistent systems are highly appropriate for high reliability applications, but, the uptake of these systems into other environments has been slow.

Many of the operating systems in table 3.1 provide memory objects as a class of items which can be directly manipulated and shared by user programs. This trend is consistent with observations that applications on conventional file based systems spend a significant amount of their time in converting file based structures to memory based structures and vice-versa. Among the file based operating systems, UNIX, has addressed this problem by providing a mechanism which allows files to mapped into a process's memory[21]. This mechanism approximates the provision of memory objects to user programs.

Many of the operating systems surveyed have implemented a mechanism which allows more than one thread of execution (commonly known as *threads*) to exist

[21]The *mmap* call was originally specified for 4.2BSD although it was not implemented in the shipped version of that release[LMKQ90]

within the address space of a process. There are two major variants. The simpler mechanism allows several threads to share the time-slices given to a process. This mechanism does not allow more than one thread to execute at a time. It interleaves the execution of the threads within the time-slice. The mechanism can be implemented at user level using upcalls to switch between threads when an event occurs (Angel uses this method) or it can be managed by the kernel (CHORUS uses this method as actors are tied to a processor and several threads may inhabit an actor). The more complex variant allows threads to execute in parallel by using different processors to execute the parallel threads. It should be noted that SASOS like Mungi and Opal do not require explicit support for threads as all processes have full access to the address space.

The thread paradigm is becoming more popular in both its implementations as it provides mechanisms that simplify the task of application programmers who wish to perform operations in parallel. The shared address space provided by threads frequently compensates for difficulty in sharing data between processes. Combining this with the lower cost of switching between threads of a single process than between processes, provides incentive for programmers to use threads to produce programs which perform actions seemingly simultaneously. The negative aspects of the use of threads is that interactions between components of a single program cease to be deterministic, and the absence of protection offered by using separate processes allows threads to readily interfere with each other.

Chapter 4

The Walnut Kernel

The Walnut Kernel intends to achieve the benefits of the Password-Capability System without requiring the special hardware used by the Password-Capability System. That is, supports the use of password-capabilities to provide protection and sharing among multiple users, object persistence, and the free mixing of capabilities with other user data, but using only the hardware likely to be available on most general purpose computers.

The survey (chapter 3) identified a number of trends in the features of operating systems. The Walnut Kernel exhibits several of these features. Few of the systems surveyed required special hardware. It became clear that to gain wide acceptance, the kernel must be supported on a wide range of platforms and must rely only on the lowest common denominator of adequate hardware.

The Walnut Kernel follows the trend towards small kernels or micro-kernels. The Walnut Kernel and the small kernel and micro-kernel systems surveyed were motivated by: reducing the size of the task of implementing the kernel of the operating system, multiple personalities, and easing the task of verification. Small kernels can be successfully implemented by small groups with limited time and resources - the Walnut Kernel was produced by one such group. The task of implementing the personalities of the operating system can be left to others outside the kernel development group as personalities are user level programs which make calls on kernel functions. The presence of multiple personalities running on a single machine is attractive as it allows users to start working in a familiar environment and migrate to the native

environment as they require the features provided by the kernel. Verification and the requirement for strong security were common themes of the implementors of the systems surveyed. As with the Password-Capability System, the security model of the Walnut Kernel is simple and can easily be shown to be probabilistically secure, providing the first step in verifying the security of the system.

Protection and security are major features of both the Walnut Kernel and the Password-Capability System. The password-capability mechanism allows the implementation of tight security and access control without imposing a hierarchy of access privileges, and without the concept of the ownership of objects. Implementors of personalities on top of the Walnut Kernel have the freedom to construct arbitrary protection mechanisms, including schemes based on ownership and hierarchies. Furthermore, the flexibility and security provided by the kernel are gained at very low cost. The kernel requires capabilities to be periodically revalidated. Revalidation is accomplished by invalidating a capability and the pages of the object at regular intervals. The capability is validated again when a page fault occurs when accessing a page of the object. This adds only a small overhead to the cost of servicing page faults when compared with other systems.

The Walnut Kernel is designed for use by most processors that support virtual memory. A parallel project in the Department of Computer Science, Monash University, was to develop hardware to support a general purpose multiprocessor which can be scaled from a single processor to a massively parallel multiprocessor. Chapter 11 describes this hardware. The Walnut Kernel and other operating systems may be supported by that hardware design.

Chapter 5

The User Perspective

This chapter provides an overview of the features of the operating system visible to a user process. Appendix A provides detailed information for writing user processes including a detailed description of the contents of **the wall** (see section 5.2), a map of the process address space, and the arguments of kernel calls. Chapter 6 provides the rationale for the design features described in this chapter.

The Walnut Kernel draws on experience gained in the Password-Capability System, and hence has adopted many of the ideas found in that system after adapting them for use in the new environment.

This chapter initially outlines the environment that is presented to a process. Subsequently, the operations available to a user process and interprocess communication are discussed.

This chapter does not contain references to input/output, files or users as these concepts are not defined at the kernel interface level.

5.1 Volumes, Objects and Capabilities

Volumes represent the physical media on which the persistent storage of the Walnut Kernel resides. The *volume number* is a unique identifier permanently associated with each physical storage device used for persistent storage by the Walnut Kernel. In addition, a *special volume* is used to represent the memory occupied by the kernel and memory-mapped interfaces to hardware devices.

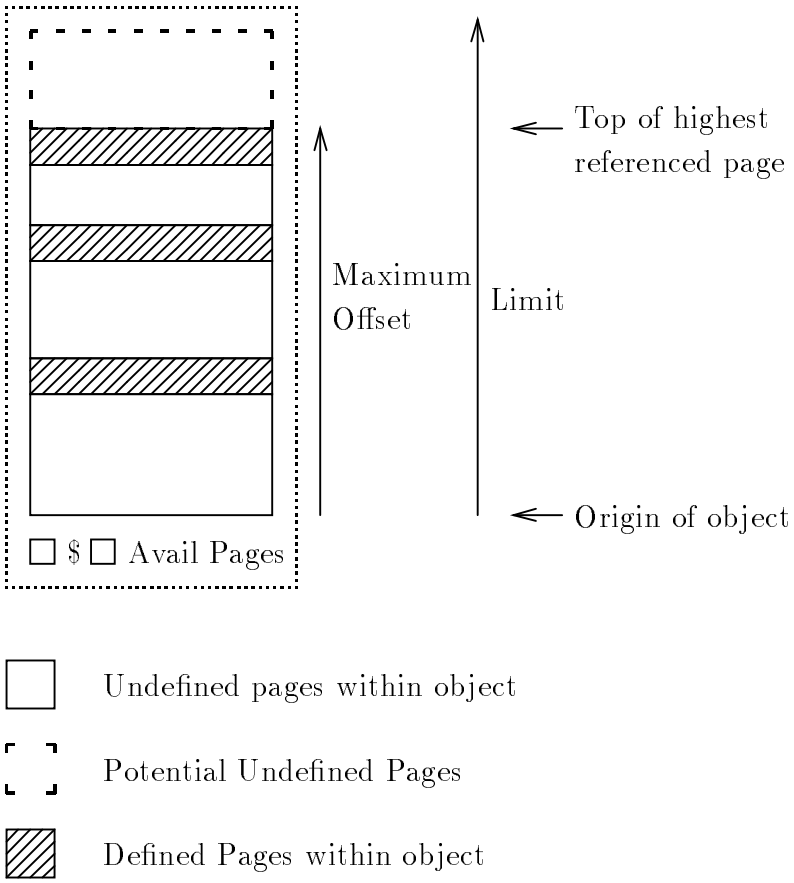


Figure 5.1: An object

Under the Walnut Kernel all the data available to user programs is stored in **objects**. These objects are analogous to segments in a paged-segmented architecture. They (see figure 5.1) consist of collections of pages defined by a maximum offset, a limit (the maximum value to which the maximum offset can be increased within the object), an amount of money, and a number of pages guaranteed to be available to it.

Objects have the following properties:

- Objects are permanently associated with a volume. Objects cannot span more than one volume, nor can they be moved between volumes.
- Pages are allocated when the first reference is made to them. To prevent data leakage, pages are blanked by writing zeros into all the locations before the page is made available to the user.
- If the number of guaranteed pages has been exceeded, and there are unreserved pages on the volume, then additional pages are allocated to the object. If there are no unreserved pages available, then an exception will occur.
- Attempts to access beyond the limit of the object will result in an exception. The limit of an object can be increased by the *resize* kernel call.
- The main memory acts as a cache of objects.

The Walnut Kernel uses Password-Capabilities (see figure 5.2) to provide naming and access control to objects. The *volume* and *serial* parts of a valid capability identify an object on a volume. Associated with each capability is a set of attributes which includes a set of rights and a view[1]. The password components are used to identify the rights the holder of the capability is allowed to exercise over the object.

[1]A view is an attribute of a capability. It defines the region of the object that can be addressed by the possessor of the capability. Views are contiguous regions and are defined by an offset from the base of the object and an extent. An important distinction from the Password-Capability System is that the view entitles the user to address part of an object, it does not guarantee that pages are contained in that region nor that the pages are readable to the user (This constraint is particularly relevant to process objects).

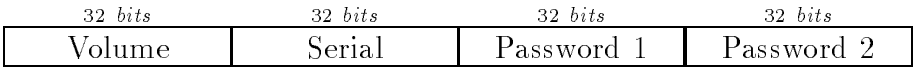


Figure 5.2: Structure of a Walnut Kernel Capability

The Walnut Kernel supports several sets of rights: system rights, user rights, drawing rights, and message rights. The **system rights** consist of a set of bits which determine whether the holder is permitted by the system to perform an operation. The system rights are listed in table 5.1. The **user rights** consist of a set of 32 bits which are managed by the kernel. The kernel attaches no meaning to the user rights bits. They are intended to be used by user processes to implement access to services in a way that is analogous to the control system rights bits have over access to kernel services. The **drawing right** of a capability is the amount of money that can be withdrawn through the capability or its descendents (see section 5.5). **Message rights** are only relevant to processes as they determine which subprocess or subprocesses a capability can be used to send a message to. There are three levels of restriction: the capability can be used to send a message to any subprocess of the process, the capability can be used to send a message to any subprocess of the process except for subprocess zero (subprocess zero is discussed in section 5.9), or the capability can be used to send a message only to a specific subprocess.

When an object is created it is allocated a master capability. If the master capability is deleted, the object is destroyed. When the master capability is created it has system and user rights specified by the creator and provides a view which covers the object. All other capabilities which refer to this object are derived from the master capability or its descendents. All capabilities referring to a given object have identical *volume* and *serial* fields. The password fields are selected randomly with the constraint that for all capabilities for an object the *Password 1* is unique.

The Walnut Kernel supports two mechanisms for generating new capabilities: making new objects and deriving capabilities from existing capabilities. The process of deriving capabilities is fundamental to the function and security of the Walnut Kernel. Derived capabilities (see figure 5.3) are made by presenting the kernel

SRDERIVE - Allow capabilities to be derived from this capability.
SRSUICIDE - Allow this capability to destroy itself and its children.
SRDEPOSIT - Allow the holder of this capability to deposit money into it.
SRWITHDRAW - Allow the holder of this capability to withdraw money from it.
SRREAD - Allow the holder of this capability to read from the section of the object covered by this capability.
SRWRITE - Allow the holder of this capability to write to the section of the object covered by this capability.
SREXECUTE - Not used.
SRUSER - Allow user processes to use this capability.
SRPEEK - Allow the holder of this capability to perform a peek system call (see A.11.2) on the process represented by this capability.
SRMULTILOAD - Allow this capability to be loaded by any process. If this right is absent then only processes with a serial number equivalent to the capability's password 2 may load this capability.

Table 5.1: System Rights

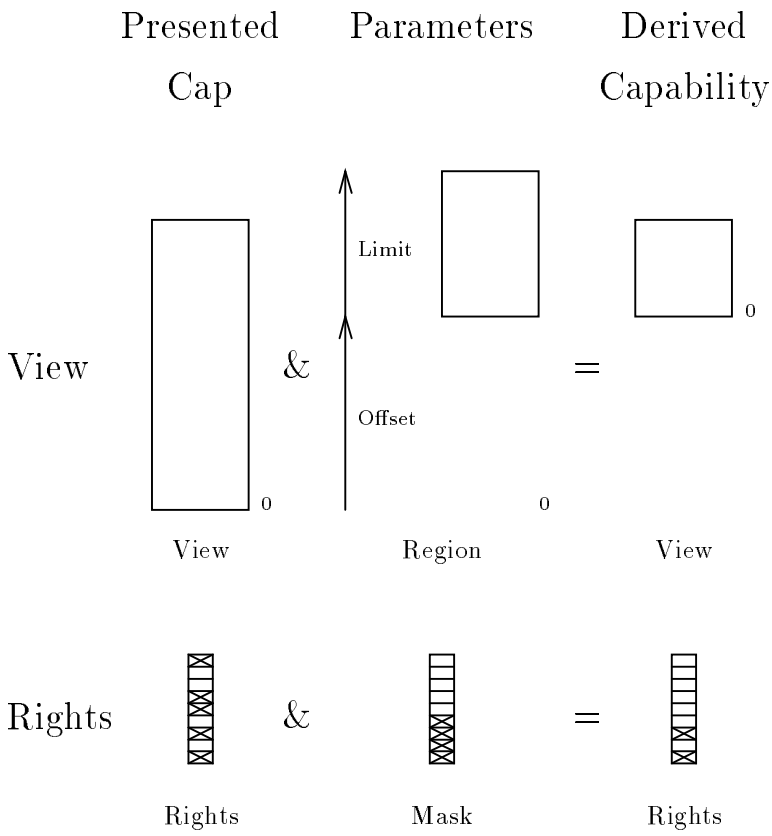


Figure 5.3: Derivation

with a **capability**, a **rights mask**, an **offset** and a **limit**, and invoking the derive operation. The system rights of the derived capability are the logical-and of the rights of the presented capability and the rights mask. The message rights may be further restricted. The drawing right is arbitrary, and may exceed that of the parent capability. The view presented by the derived capability is the region of the object that is covered by the intersection of the view of the presented capability and the region covered by the offset and limit.

Derived capabilities, at the time of derivation, have equal or lesser rights than than their parent capability. *Suicide right* is an exception as this right may be added to the children of capabilities which do not hold this right.

The rights of a parent capability may be reduced through the use of the *restrict* kernel call after a child capability has been derived. The child capability is unaffected by the restriction of the parent capabilities rights.

Each object is assigned an object type[2] when it is created. The creator may choose any 32 bit value for an object type with the exception of a few system reserved values and the most significant bit of the type. The most significant bit of the object type is set for process objects and clear for other objects.

Process objects are distinguished from other objects in the system as part of the contents are interpreted by the operating system. The process object's major properties are:

- it defines an address space in which processes operate, by a Table of Loaded Capabilities (TLC) which is contained in the process object

- it contains the state of 1 or more subprocesses

Although the master capability of a process allows all the contents of the process object to be addressed, parts of the process object are read-only and other parts permit no attempts at access (see section 5.2).

[2]An object type is a 32 bit value that can be accessed by holders of an object's capability

5.2 Process Address Space

The address space seen by a running process is defined by the contents of the Table of Loaded Capabilities (TLC). This table consists of a list of capabilities and their mappings into windows. The mapping describes the offset from the start of the object, the extent of the addressable region, and the access rights conveyed by the capability. The address space of a Walnut Kernel process (figure 5.4) consists of a number of regions.

The **kernel area** is located at the low end of the process's address space (`0x0` to `0x3fffff`) and is neither readable nor writable by processes. The **wall**, a single page located at `0xc000`, is an exception as it can be read by processes.

The **wall** is a page mapped into all processes which contains public information. This information includes the capabilities of utilities, resources and the current time. Processes known as wall managers can update the contents of the wall.

The **small window area** (`0x400000` to `0xffffff`) is located above the kernel area. This area can be loaded with views of objects which start and end on arbitrary page boundaries.

The **process object** is loaded at `0x1000000` and extends through to `0x3ffffff`. Figure 5.5 is a map of the process object. The **process header** is not accessible to processes. The **address map** contains information that allows *capability indices* to be translated to and from locations in the process address space. It is typically used to translate addresses to *capability indices*. The **parameter page** consists of the **parameter block** and the **message area**. It is used to transfer information when kernel calls are made. The remainder of the process object can be used for the storage of data or code.

The **large window area** extends from location `0x4000000` to the top of the system memory. This area can be loaded with views of objects which start on offsets exactly divisible by `0x4000000` and, either end on an offset exactly divisible by `0x4000000`, or end on the end of the object.

The Walnut Kernel allows up to 250 views of objects to be loaded into a process's address space.

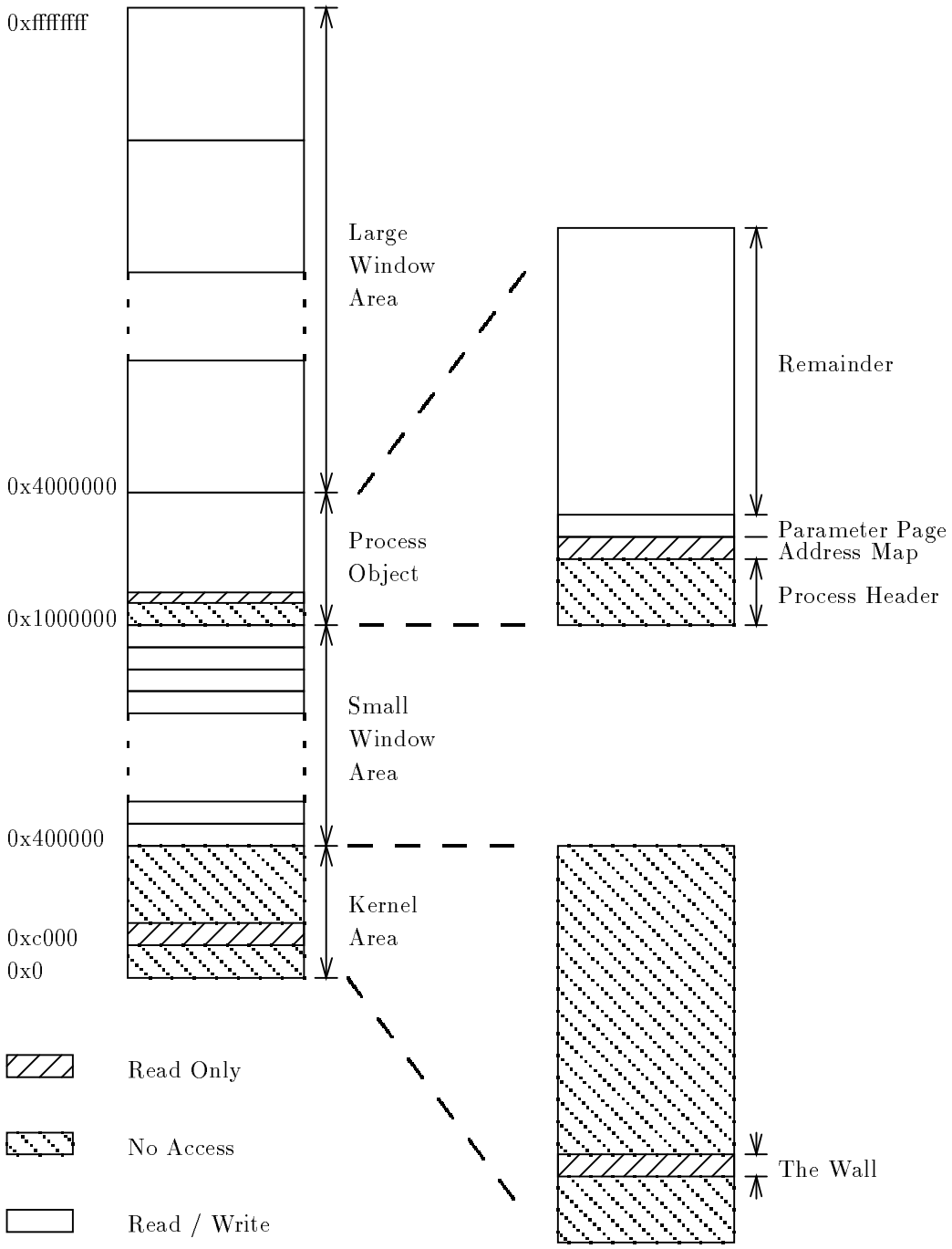


Figure 5.4: Process Address Space

Region	Start - End	Access
Process Header	0x1000000-0x100efff	No Access
Address Map	0x100f000-0x100ffff	Read Only
Parameter Page	0x1010000-0x1010fff	Read / Write
Parameter Block	0x1010000-0x101004b	Read / Write
Message Area	0x101004c-0x1010fff	Read / Write
Remainder	0x1011000-0x3ffffff	Read / Write

Figure 5.5: Map of the Process Object

5.3 Processes and Subprocesses

A process defines an environment in which subprocesses operate. The environment consists of an address space (specified in section 5.2) and a collection of resources.

When a process is created the maximum number of subprocesses and the number of mailboxes are set. The number of these process resources cannot be varied during the existence of the process. Two subprocesses are created when a process is created: subprocess zero (subprocess zero is discussed in section 5.9) and subprocess one.

Subprocess one executes user code. It, or its descendents, can create other subprocesses, all of which execute code that has been mapped into the process address space. Subprocesses have no protection from the actions of other subprocesses within the same process.

Every subprocess has a priority in the range of 0 to 254 and a wakeup time. Subprocess zero has a unique priority of 255, the highest possible. The wakeup time is a clock time in seconds. A subprocess will not run until the system clock equals or exceeds its wakeup time.

The scheduling of subprocesses is similar to the scheduling of processes on a single processor time sharing system. When a subprocess of a process is executing, no other subprocess of that process can be executing. The algorithm for determining which subprocess of a process to execute in the current time-slice is as follows:

1. If a subprocess is executing and there is a non-zero value in the *reserve* field of the parameter block, resume execution of that subprocess.

2. Execute the subprocess with the highest priority which is not waiting.

3. For subprocesses of equal priority, select the first subprocess encountered in the subprocess table.

5.4 Messages and Mailboxes

Each process has a number of mailboxes allocated when it is created. Mailboxes may be configured to reject all mail, or accept messages addressed to a specified subprocess and/or messages starting with a specified string. The application programmer can guarantee the availability of suitable mailboxes for a given type of message, provided the application is not already holding pending messages of that type.

When a process is created a mailbox is opened for subprocess zero. All the other mailboxes of the process are initially closed, until the process explicitly opens them. This allows the process to perform its initialization before starting to handle messages.

Messages perform the following functions under the Walnut Kernel:

- Data Delivery - up to 16 words of information can be transfered to the destination process in the body of a message.

- Money - messages can deliver money to a process. Attached to each message is a sum of money which is transfered to the destination process when the message is stored in a mailbox. Negative sums of money are not permitted.

- Scheduling Control - when a message is delivered it wakes up a sleeping process and makes the message's target subprocess runnable by resetting its wakeup time.

Messages and subprocesses provide a mechanism for encapsulating events which are not synchronized with the operation of a process.

The presence of a message in a mailbox prevents a process from waiting.

There are two ways of directing messages to specific subprocesses:

- Using a capability that is restricted to send to only one subprocess of a process ensures that any message sent with that capability will only be delivered to the subprocess named by the capability.

- A subprocess must be specified as a parameter of a **send** operation when a capability which is not restricted to a single subprocess is employed.

The first mechanism is typically employed by server processes giving capabilities to processes which will be making use of the server's facilities. This mechanism ensures that messages cannot be delivered to other subprocesses of the server, and this limits the opportunity for both error and malicious attack. The second mechanism is typically employed by holders of a process's master capability.

Mailboxes may be set up to admit only specified types of messages. This selective acceptance of messages was introduced to ensure that important messages could be guaranteed delivery even if all other mailboxes were full. Mailboxes can be reserved according to the following criteria:

- Subprocess number: A mailbox can be reserved so that it will only accept messages addressed to a specified subprocess.

- Message prefix: A mailbox can be reserved so that it will only accept messages that begin with a specified string.

- Subprocess number and message prefix: only messages addressed to a specified subprocess and beginning with the specified prefix are accepted.

When a message is sent to a process it is placed in the first mailbox found that will accept the message. There is no mechanism which can be used to control the order of filling mailboxes.

A subprocess can determine the order in which messages are retrieved. The **receive** operation allows the subprocess to specify a message prefix. The prefix is used to retrieve the first message starting with a matching string addressed to the subprocess.

5.5 Money

The Password-Capability System introduced the concept of rental - for garbage collection - and extended the use of money throughout the system to provide a system-wide economy. The Walnut Kernel adopted the economic model used in the Password-Capability System. This section describes the manipulation of money in the Walnut Kernel.

Each object has a **money word** which stores the amount of money available to the object. The money word performs two tasks: it acts as a store of money accessible to the holders of capabilities with withdrawal rights, and provides funds to pay for the rental of the disk space used by the object. Processes have an additional store of money known as the **cash word**. The money stored in the cash word is used to pay for kernel services, and acts as storage for money transfered by a process to and from objects. The kernel performs all operations which manipulate the transfer or use of money.

Associated with each capability is a **drawing right**. The drawing right of the master capability is synonymous with the object's money word. The drawing right determines the amount of money that can be withdrawn through a capability. Capabilities with a drawing right of zero cannot be used to withdraw money from the object to which the capability refers.

Two operations are supported on drawing rights: deposit and withdrawal. These operations are applied to the drawing rights of all the ancestors of the capability which is named by the operation. Withdrawal is only permitted when the drawing rights of all the ancestors and the capability itself are greater than the amount to be withdrawn.

Figure 5.6 is used to illustrate the effect of deposit and withdrawal operations. Monetary operations affect all the drawing rights on the path to the root of the capability tree for an object. If a deposit is made via capability $C^{3,2}$ then the drawing rights $D^{3,2}$ and D^3 and the money word - M - are increased by the amount of the deposit. To make a successful withdrawal from $C^{3,1}$ then $D^{3,1}$, D^3, and M must be greater than the amount to be withdrawn. If this condition is met then $D^{3,1}$, D^3, and M will each be debited the amount to be withdrawn.

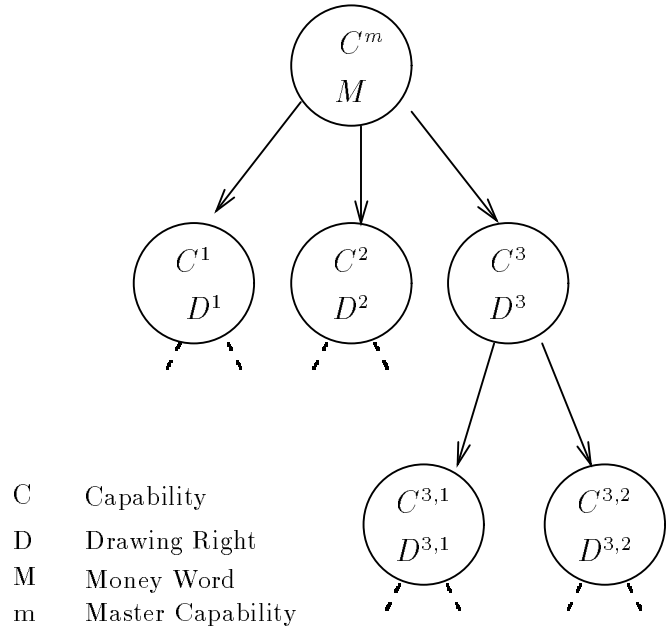


Figure 5.6: A Tree of Capabilities

5.6 Kernel Calls

The parameters used by a **kernel call** are drawn from the parameter page. The results of a kernel call are returned in the parameter page. The parameter block is a data structure located at the beginning of the parameter page. The fields of the parameter block are listed in figure 5.7.

The **reserve** field is used to ensure that values of the parameter block are not corrupted by other subprocesses of the process. When the **reserve** field is set to indicate the required kernel call, other subprocesses are prevented from being scheduled until the reserve field is set to zero[3]

To make a kernel call, the reserve field is set to the kernel call type, the required fields of the parameter block are set, data is placed in the message area, if required, and a system call is issued. The process blocks until the kernel call is completed.

On return from a system call the **error** field contains either zero or an error code. If **error** is zero, the kernel call completed successfully, and information can

[3]Section 6.6.1 describes the implementation of the *System-Call interface* and the rationale for disallowing concurrent system calls by multiple subprocesses.

error	returned error code
vol	volume number
serial	serial number
pass1	password 1
pass2	password 2
srights	system rights
urights	user rights
base	offset of capability from front of object
limit	max allowed offset from base
money	money to be transfered
type	object type
maxoff	max offset used / requested
maxsz	max size of defined content
maxcap	max capabilities now allowed
offset	offset into a capability window
subpn	subprocess number
cindex	index of capability in the table of loaded capabilities
clocktime	time in seconds
reserve	non-zero value reserves for sub-process and identifies kernel call

Figure 5.7: Structure of Parameter Block

be recovered from the fields of the parameter block and the message area as required. When all the required data has been extracted from the parameter page, it is necessary for the program to write a zero into the **reserve** field to allow other subprocesses of the process to be scheduled.

Kernel calls always return. If the call requests an illegal operation, including references to undefined addresses, the call returns with an error value in the **error** field of the parameter block.

5.7 Exceptions

Events which raise processor exceptions can be managed through the use of trap handling subprocesses. Each subprocess of a process may have a specific trap handler associated with it. A process can register any subprocess other than subprocess zero or the subprocess itself, as a trap handler for a subprocess of the process. The trap-handler is invoked when a processor exception is raised. Exceptions are grouped to

allow for processor independent trap handlers to be written.

Five types of exception can occur. A *floating point* fault is used for all arithmetic errors, including underflow and division by zero. *Opcode* faults are raised when illegal opcodes are detected. An *address* fault is raised when a memory access violation occurs. *Debug* faults are system dependent and are used to implement debuggers. *Alignment* faults are raised on non-aligned accesses when the processor detects this type of error.

If a trap handling subprocess has not been assigned, then the default action is to terminate the process.

When an exception is raised, the faulting subprocess is made unrunnable and a message is sent to the exception handling subprocess for the faulting subprocess. If the message is undeliverable the process is terminated.

5.8 Controlling Process Scheduling

The Walnut Kernel provides two mechanisms for controlling the scheduling of processes:

- **Wait** and **Messages**
- **Freeze** and **Thaw**

This section describes the conventional mechanism which employs the wait system call and messages. The freeze and thaw mechanism is accessed through subprocess zero, and is discussed in section 5.9.

The **wait system call** is used for several purposes:

- An argument of -1 to the system call sets the running subprocess's wakeup time to **forever**[4] and causes the scheduler to remove the process from the scheduling queue if there are no other runnable subprocesses of the process.
- When the argument is 0, the system call surrenders the remainder of a process's time-slice.

[4]Forever is defined as `0xffffffff`. It is the largest value that the clock can represent.

- When any other argument value is used, then the wakeup time of the subprocess is set to the value of the argument

The wait call is used to put subprocesses to sleep when they do not have useful work to perform.

The presence of a message causes a subprocess's wakeup time to be set to now. This ensures that processes with waiting messages are scheduled.

The arrival of a message causes a subprocess's wakeup time to be set to zero and the process to be placed into the scheduling queue. It does not cause the receiving subprocess to pre-empt any other process.

A common construct in server processes is the message loop. This construct may be represented in pseudocode as[5]:

```
while true
begin
    wait(-1)
    receive(msg)
    server_function(msg)
end
```

The message loop waits until a message is present after which it performs a task which acts on the message. If a message should arrive while the current message is being acted on, the wait operation will have no effect in the next cycle of the loop. The second message can be handled immediately after the first message has been handled.

The wait and message mechanisms can be used to implement sychronisation operations. In this application one subprocess waits, and remains blocked until it receives a message which allows the process to continue. It is necessary to issue a receive call after the wait as the presence of the message will prevent further wait operations having effect.

[5]The kernel operations of receive and wait have been encapsulated in subroutines to simplify the code

5.9 Subprocess Zero

Subprocess zero is part of the kernel. It performs tasks in response to messages which request services. These tasks typically change the state of the process - for example, one of the functions shifts the process from running to suspended - or report the state of the process.

A mailbox is opened for subprocess zero when a process is created. Subprocess zero's reserved mailbox is never closed while the process exists. This arrangement ensures that the creator of a process has control over the process at all times.

Subprocess zero currently supports the following functions:

Freeze prevents a process from being scheduled. On receipt of a freeze message subprocess zero sets the process state to frozen, and causes the process to be removed from the scheduler queue.

Thaw allows a process to be scheduled. When a process receives a message it is placed into the scheduler queue. If the process is frozen, the process is typically removed from the queue after the subprocess zero messages are parsed. On receipt of a thaw message, subprocess zero sets the process state to normal and process execution resumes.

Wakeup sets the wakeup time of the specified subprocess to now. The wakeup message sets the wakeup time of the nominated subprocess to the current time. The wakeup message is used to start a process that has suspended activity and has closed mail boxes. It relies on the fact that the mail box allocated to subprocess zero cannot be closed.

Cooee requests the process to send a status message using a specified capability. The reply message sent by subprocess zero consists of a set of words which represent the Cooee reply identifier, the volume and serial number of the current process, and a process status.

Protected Freeze prevents a process from being scheduled until all protected freezes on the process have been thawed. On receipt of a protected freeze message subprocess zero sets the process state to frozen, XORs the magic

word contained in the message with a key held in the process state, increments a count held in the process state and causes the process to be removed from the scheduler queue. The XOR operation and the count prevent other parties from thawing the process unless they know the set of magic words used in the protected freeze operations applied to the process.

Protected Thaw allows a process to be scheduled when all other protected freezes have been thawed. On receipt of a protected thaw message subprocess zero XORs the magic word contained in the message with a key held in the process state and decrements a count held in the process state. If both the count and key held in the process state are zero, then the process is thawed. If the count is zero and the key is non-zero then the process is terminated.

Chapter 6

Design of the Walnut Kernel

This chapter describes the guiding principles in the design of the Walnut Kernel, its overall architecture, the rationale of some key decisions, and its detailed structure and operation.

6.1 Design Principles

The following design principles guided the development of the Walnut Kernel:

- Avoid features available only on small classes of processors
- Avoid stalls in the kernel while waiting on external events
- Minimize retained kernel state variables
- Ensure the kernel is scalable
- Use static allocation of kernel memory
- Allow for a variety of shared memory architectures

The principles, the motivation for their inclusion as design principles and their implications are discussed in the remainder of the section.

6.1.1 Avoid features available only on small classes of processors

The original Monash Multiprocessor Project used purpose built hardware to assist with the management and use of capabilities. A consequence of this decision was that the Monash Multiprocessor Project was tied to a specific processor, memory architecture, bus architecture and implementation of these. This design placed an upper bound on the performance of the system and, combined with the long lead times and expense of developing experimental hardware, prevented the system from advancing. With improvements in technology the advantages conferred by the design of the hardware were out-weighed by the advantages offered by the improved technology. The system fell into disuse when it became clear that equivalent performance could be gained using conventional technology.

By designing the Walnut Kernel to operate on a wide range of processors and by placing minimal requirements on the type of memory management expected by the kernel to be available to the processor, it is hoped that Walnut Kernel will be less prone to obsolescence caused by improvements in available hardware.

This principle influenced both the design and implementation levels. An example at the design level is the requirements on page table or translation-lookaside buffer entries; the Walnut Kernel expects the presence of *valid* and *dirty* bits, but does not expect (or use) *use* bits. Although the majority[1] of processors support both *use* and *dirty* bits neglecting the presence of *use* bits caused no loss of performance or utility. An implementation lacking a *dirty* bit would have either a significantly increased number of page faults or a significantly increased number of writes to disk, decreasing system performance. The design opted to neglect the presence of *use* bits but require the presence of *dirty* bits.

A direct consequence of this design principle is that the Walnut Kernel does not take advantage of the segment registers available in the Intel386[2]. The segment registers on the Intel386 would permit objects and views with byte or word size

[1] The VAX architecture provides only a *modify* bit (dirty bit) [Cor86].

[2] Intel386 is a trademark of Intel Corporation

granularities to be implemented. However, as the Intel386 / i486[3] is one of the few families of current microprocessors supporting segment registers and the trend in microprocessor design is away from the use of segmentation, it was decided to use the more generally available mechanism of paging.

6.1.2 Avoid stalls in the kernel while waiting on external events

The kernel and drivers are implemented with the policy:

> Upon encountering a state that would cause the kernel to stall or wait before being able to continue, the kernel will initiate a corrective action and then proceed with another task.

A consequence of this policy is that the kernel and drivers avoid tight busy-waiting loops on external events. This policy avoids the catastrophic consequences for system performance that result from uninteruptable loops waiting on delayed events. The policy is particularly applicable in a multiprocessor environment where actions with other processors are not synchronized and the competition for access to a resource may take a significant amount of time. The policy allows the kernel to do other tasks if the current task is prevented from making immediate progress.

With suitable hardware assistance the Walnut Kernel can be configured to allow for long propagation times. A DMA like operation could be initiated to lock and modify data. By surrendering control from the current task and scheduling another task the kernel can continue performing local operations while the operation with a long propagation delay is completed by the hardware.

6.1.3 Minimize retained kernel state variables

When any kernel activity is invoked, for any reason, the kernel will attempt to effect some change in the system state. Whether or not it succeeds in completing this change, the system state is left in a consistent configuration independent of any

[3] i486 is a trademark of Intel Corporation

local variables of kernel routines. Thus there is no need for retention of kernel state information and no need to support multiple threads of activity within the kernel.

The kernel is not regarded as a process with continuing state and threads of execution. Rather, it is regarded as a set of state transformation rules. A rule (kernel action), once invoked, either runs to completion or is abandoned with the transformation incomplete because of the need for a disk transfer, or a resource conflict. An abandoned rule leaves the system in a state where the transformation can be later completed.

6.1.4 Ensure the kernel is scalable

The kernel was designed to achieve decentralized operation with additional kernels to run in parallel. The absence of centralized control allows for easy scaling in terms of numbers of kernels running. Decentralized operation also encourages fault tolerance at the kernel level.

With suitable hardware and the *correct-and-retry* nature of the Walnut Kernel it is possible to reduce the effects of long propagation delays for remote data accesses on the throughput of the system.

6.1.5 Allow for a variety of shared memory architectures

There are two broad categories of multiprocessor memory architecture: shared memory or discrete memory. In shared memory systems all the processors have access to the same information in memory at the same time. The hardware provides support for coherence. In discrete memory systems the kernel must provide mechanisms allowing access to pages of memory held by other processors. Typically the mechanisms involve copying pages to local memory and providing many-readers-single-writer access to the pages. The Walnut Kernel is designed to operate with both these architectures or their hybrids.

It is necessary to support both categories of memory architecture as it affects the scalability of the system. The general trend is towards using symmetric multiprocessors (shared memory) for small systems and towards discrete memory architectures for large multiprocessors. Adding processors to a symmetric multiprocessor is cost

efficient while the memory bus is not saturated as only the processor module needs to be duplicated. As the memory bus approaches saturation the potential return of adding additional processors to this shared bus architecture decreases. For systems with a larger number of processors the bandwidth of the bus and shared memory eventually ceases to be able to meet the demands of the processors. To avoid this problem it is typical to have separate buses and memory modules with some communication network linking the buses and memory modules. Accordingly a discrete memory structure is usual for larger multiprocessors.

6.2 Passive Elements

6.2.1 Disk Structures

Objects are the basic unit of the Walnut Kernel. An object is composed of a *body* and a *dope*. The *body* of an object consists of an array of bytes. Not all of the bytes of an object are necessarily defined. All the system information about an object is stored in the object's *dope*. The inclusion of all the system information as part of the object is a distinguishing feature of the Walnut Kernel.

An object resides completely on a single *volume*. Each volume has an identifier permanently associated with it known as the *volume number*. Volumes are usually random-access, block-oriented storage devices. The most common devices used for volumes are disks.

An object is divided into *pages*. The size of a *page* is a multiple of the page size of the host processor. Data is transfered to volumes in units of *blocks*. A block occupies a logically contiguous section of a volume. *Blocks* and *pages* are defined to be of the same size.

The *blocks* of a volume are numbered from zero to the number of blocks on a volume minus one. The number of a *block* is known as its *location* on a volume.

Both the *body* and the *dope* of an object are stored as sets of blocks on a volume. The sets are not required to be contiguous.

The body of an object has no system-imposed structure unless it contains a process.

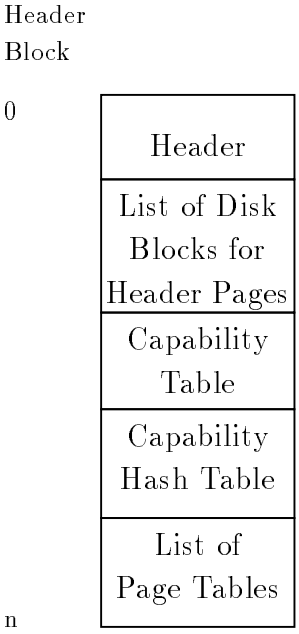


Figure 6.1: Object Header Blocks / Pages

The *dope* consists of two structural elements: the *object header* and a set of page tables. The *object header* consists of one or more blocks. Figure 6.1 shows the contents of *header blocks* of an object when joined in order. The *first-header-block* starts with the structure **Header** (see figure 6.2). A list of the locations of disk blocks containing the *header blocks* of the object is stored after the **Header** data structure. The *capability table* lists all the capabilities for the object. Empty slots in the *capability table* are linked together to form a free list. The *capability-hash-table* is an index into the *capability table* based on the first password of a capability. A list of the locations on disk of the page tables of the object is the final element of the *header blocks*.

The *first-header-block* contains sufficient information to retrieve the remaining *header blocks* and hence allow access to all the pages of an object. The placement of the list of *header blocks* ensures that the reference to the second *header block* - if required - occurs within the first block guaranteeing access to the second and subsequent header blocks. As the header blocks contain references to the location of page tables which contain references to the location of pages of an object, all the pages of an object can be located using the header blocks of an object. Both pages

and page tables may be undefined for parts of an object which have not yet been accessed.

The structure **Header** (see figure 6.2) begins with a magic number that identifies the object as the *first-header-block* of an object. The **dopesz** field contains the number of bytes of header information. The **type** field is a 32 bit object type identifier. The top bit of the **type** field is set for process-objects and clear for data-objects. The master capability of the object is stored in the **vol**, **serial**, **pass1**, **pass2**, **base**, **limit**, **money**, **srights**, and **urights** fields. The information relating to the master capability and the **linc** field forms an entry in the *capability table*. The **dopeblks** field contains the number of header pages used by the object. The **maxsz** field contains the number of bytes guaranteed to be available to an object to store header blocks and data blocks. The **maxoff** field contains the largest addressed offset into an object. The **maxpage** field contains the highest addressed defined data block in the object. The size of the table available for holding capabilities is stored in **maxcap**. The **numcap** field contains the number of capabilities stored in the table. The **hashmsk** field is equal to the index of the top element in the *capability-hash-table*. The **freeindx** field contains the index of the first element of the free list for the *capability table*. The **maxtabs** field contains the maximum number of page tables required for **maxoff**. The **totdef** field contains the number of disk blocks allocated to the object. The **maxdef** field contains the number of pages needed for **maxsz**. The **dlocoff**, **hashoff**, **capoff**, and **taboff** fields respectively contain the byte offset of the list of disk blocks of header pages, the *capability table*, the *capability-hash-table*, and the list of disk locations of page tables. The **dreftime** and **altime** fields contain the last reference time and the last alter time of the object. The **squeeze** field is used to indicate that the page table of a small object has been squeezed into the header pages of an object to conserve space. The **link**, **state**, and **restabs**, fields are ignored when the *object header* is on disk.

The low order bits of the *serial number* of an object contain the *block number* of the *first-header-block* of the object. The high order bits of the serial number are randomly selected when an object is created.

The construction of the serial number of an object is a key feature of the design

```
typedef struct Headerst {
        Uw magic;
        Sw dopesz;
        Sw link;
        Uw type;
        Uw state;
        Uw vol;
        Uw serial;
        Uw pass1;
        Uw pass2;
        Sw base;
        Sw limit;
        Sw money;
        Uw srights;
        Uw urights;
        Sw linc;
        Sw dopeblks;
        Sw dlocoff;
        Sw maxsz;
        Sw maxoff;
        Sw maxpage;
        Sw toppage;
        Sw maxcap;
        Sw numcap;
        Sw hashmsk;
        Sw freeindx;
        Sw capoff;
        Sw hashoff;
        Sw maxtabs;
        Sw restabs;
        Sw totdef;
        Sw maxdef;
        Sw taboff;
        Uw dreftime;
        Uw altime;
        Sq squeeze;
        Sq dum1;
        Sq dum2;
        Sq dum3;
        } Header;
```

Figure 6.2: The **Header** Data Structure

of the Walnut Kernel as it eliminates the need for a catalog of objects on a volume. However, the kernel needs to be able to distinguish between an ordinary data block and the *first-header-block* of an object to prevent users creating a block with the format of a *first-header-block* and using the fake header block to access the pages of other objects. A *bitmap* has been introduced to identify the contents of disk blocks.

The *bitmap* consists of a contiguous set of disk blocks. The *bitmap* provides a two bit summary of the usage of every disk block on a volume. Blocks are **free**, the **first-header-block** of an object, **in-use**, or **bad**. Free blocks are available to be allocated by the kernel. Bad blocks are ignored by the kernel. The *first-header-block* of an object is distinguished from other allocated blocks that are currently in use.

The essential contents of a *volume* are a *Disk-ID-block* and a *bitmap*. The *Disk-ID-block* identifies the locations of key data structures on a volume, and the logical and physical details of the volume.

The first word of the *Disk-ID-block* contains a bit pattern which identifies the block as a *Disk-ID-block*. If the signature is incorrect, the volume is assumed to be either corrupt or invalid. The remaining words contain: the total number of blocks on the disk, the number of used blocks, the index of the first bitmap block, the number of available blocks on the disk, the **Disk-Block-Mask**, the name of the volume, the serial number of the initialization process (see section 6.8), and the number of reserved blocks on the disk.

```
typedef struct Captabentst {
        Uw pass1;
        Uw pass2;
        Sw base;
        Sw limit;
        Sw money;
        Uw srights;
        Uw urights;
        Sw link;
        Uw dad;
        } Captabent;
```

Figure 6.3: The **Captabent** Data Structure

The *capability table* is built from **Captabent** structures (see figure 6.3). The **pass1** and **pass2** entries contain the passwords of the capability. The **offset** field

gives the offset of the capability window from the base of the object. The **limit** field contains the size of the capability. The **money** field contains the *drawing right* of the capability. The *drawing right* of the master capability is known as the money word and represents the money held by the object. The **srights** and **urights** fields contain the system and user rights information for the capability. The **dad** field contains the index of the parent of the capability. The **link** field points to the next element in the hash chain of capabilities.

Figure 5.1 in chapter 5 identified three parameters used to describe the space allocation of an object: the *maximum offset*, the limit, and the number of pages allocated to an object. The three parameters correspond to the **maxoff**, **maxsz**, and **limit** fields in the **Header** of an object. The parameters have been selected because they allow efficient sizing of tables within the *object header*. The **maxsz** parameter determines the number of blocks reserved for the use of an object on a volume. Reserving disk blocks prevents the over committing of resources on a volume[4]. The **maxoff** field determines the number of page tables required at present. The **limit** field determines the number of page tables allowed for the object.

6.2.2 Memory Structures

The *Volume Table* lists the volumes that the kernel can access. It is constructed from an array of **VolTabEnt** structures. The **VolTabEnt** structure (see figure 6.4) holds a pointer to the queue used by the kernel to communicate with the device driver which manages the volume. The entry contains all the physical information required to access the disk including: the device type (**devtype**), a pointer to a device properties structure containing information found out about a device at boot time (**physchar**), the number of reserved blocks (**reserv_blocks**), the locations of the *Disk-ID-Blocks* (**idblk1** and **idblk2**), the location of the disk's Bitmap (**map_block**), the size of the disk (**size**), the mask used to extract block locations from serial numbers (**dblkmsk** and **ndblkmsk**), the number of available blocks (**avail**), and the number of used blocks (**used**). Pointers to copies of the *Disk-ID-*

[4]Media failures can result in reserved blocks being unavailable, however, it is not practical to guard against all hardware failures

```
typedef struct VolTabEnt {
        Uw vol;
        struct disk_q_head *queue;
        Uh status;
        Uh devtype;
        struct DevProp *physchar;
        Uw reserv_blocks;
        Discmapent *map;
        Uw *idblk;
        Uw idblk1;
        Uw idblk2;
        Uw max_vol_size;
        Uw map_block;
        Sw size;
        Sw dblkmsk;
        Sw ndblkmsk;
        Sw avail;
        Sw used;
        Uq lock;
        Uq pad1, pad2, pad3;
        } Voltabent;
```

Figure 6.4: The **VolTabEnt** Data Structure

Block and the Bitmap which are stored in memory are held in the **idblk** and **map** fields.

The *Active Object Table (AOT)* holds information about each object loaded into memory. The *AOT* contains memory images of object headers (the header blocks of an object: see figure 6.1). The majority of the fields of the object header are identical when the page is in memory or on disk. However, the **link**, **state**, and **restabs** fields have meaning when the object header is loaded into the *Active Object Table*.

The *AOT* is managed using a heap discipline. Information relating to a particular object is found by using a hash table. The serial number of a capability is used as key into the *aothash* table. The hash table contains offsets into the AOT and **Header** data structures are linked together to form hash chains. The **link** field is used to point to the next element of the hash chain.

The contents of the **state** field indicates the type of change the object is undergoing. An object can be accessed freely if it is in normal (**DOPENORMAL**) state.

An object is completely inaccessible while the header blocks are being brought into memory (**DOPECOMING**) or removed from memory (**DOPEGOING**), the capability list is being compacted (**DOPECLEANCAP**), the object is being resized (**DOPERESIZE**), or the capability list is being rehashed (**DOPEREHASH**). Object headers are marked **DOPEDYING** if the object is being destroyed, or **DOPEMAKING** if the object is being made.

The entries in the *aotlock* table correspond to the entries in the *aothash* table. The *aotlock* table provides locks for each hash chain in the *AOT*. A hash chain is locked whenever an alteration is made to an *object header*, a page table, or the *physical memory table*. The design of the kernel guarantees that the locks are held only for short time. Centralising the locks in the *aotlock* table removes the need for locks in many of the other kernel data structures.

The Walnut Kernel uses demand paging with two levels of page tables. The top level page tables are associated with a process. Second level page tables are associated with objects[5]. A page table entry may contain the location of a disk block on a volume, or a reference to a page in memory and its associated permissions. The kernel must be able to distinguish between entries containing disk locations and those containing memory locations. Two bits are required to identify entries that contain disk locations as an entry may contain a memory location, yet be invalid. Entries which refer to disk locations have a clear **PTEPRESENT** bit (clear *valid* bit) and a set **PTEDISC** bit.

The *Physical Memory Table (PMT)* holds information relating to the state of each page frame of the memory. Figure 6.5 illustrates the two types of entries found in the Physical Memory Table. Both types of *PMT* entries identify the object from which the page was drawn by volume (**vol**) and serial number (**serial**). The **ventry** field is the index into the **volume table** for **vol**. The **dblk** field contains the location on disk for the page. The **type** field identifies how the page is used. Pages may be marked as kernel pages (**FRAMEKER-**

[5]Second level page tables may also be associated with processes. When a second level page table is associated with a process it is known as a *Private Page Table (PPT)*. Small windows are implemented using *PPT*s. For simplicity the discussion of second level page tables associated with processes is deferred until small windows are covered (see section 6.5).

```
typedef struct Fizentst {
        Uw vol;
        Sw ventry;
        Uw serial;
        Sw pagenum;
        Sw dblk;
        Uw *ref;
        Uw dumw;
        Uq type;
        Uq state;
        Uq dum2;
        Uq dum3;
        } Fizent;

typedef struct Fizenttst {
        Uw vol;
        Sw ventry;
        Uw serial;
        Sw tablenum;
        Uw dblk;
        Sw respages;
        Uw freftime;
        Uq type;
        Uq state;
        Uq dirty;
        Uq dum3;
        } Fizentt;
```

Figure 6.5: The **Fizent Fizentt** Data Structure

NEL), part of the AOT (**FRAMEDOPEBUFFER**), second level page tables (**FRAMEPAGETABLE**), data pages (**FRAMENORMAL**), top level page tables (**FRAMEDIRECTORY**), private page tables (**FRAMEPPT**), or uncached pages used for DMA buffers (**FRAMEIOBUF**). The **state** field is used to indicate progress in bringing in and removing pages from memory.

The **Fizent** data structure describes data pages. The **ref** field points to the page table word which corresponds to the physical page represented by the entry in the *PMT*.

The **Fizentt** data structure describes page tables. The number of memory resident pages for the page table is stored in **respages**. The last time a page from the table was referenced is held in **freftime**. Allocation or deallocation of a disk block in the body of an object results in an alteration to a page table. The **dirty** field is used to indicate if a page table has been altered and should be written to disk.

6.2.3 Processes

Processes are the animate elements of the Walnut Kernel. A *process object* is an object which contains the state of a process. The most significant bit of the type field of an object is set to indicate that an object is a *process object*. The data structures holding the state of the process are stored at known offsets. In general, the pages holding the process state are not readable by user processes.

The **Prochd** data structure (figure 6.6) is stored in in the first data block of the process object. The **master** field holds a copy of the master capability for the process. The **nessp** field holds the the number of *necessary pages* for the process. All the *necessary pages* of a process must be loaded into memory before a process can be scheduled. The **state** field indicates the state of a process. A process may be runnable (**PROCSTATENORMAL**), performing a system-call (**PROCSTATEKERNEL**), handling a page fault (**PROCSTATERFAULT** for read faults or **PROCSTATEWFAULT** for write faults), frozen (**PROCSTATEFROZEN**), performing house keeping tasks after it has died (**PROCSTATEPROBATE**), dead (**PROCSTATEDEAD**), or protected frozen (**PROCSTATEPFROZEN**). In addition if it is not currently in the scheduling queue it is

```
typedef struct Prochdst {
        Capl master;
        Uq nessp;
        Uq state;
        Uq action;
        Uq stage;
        Uq currsubp;
        Uq maxsubp;
        Uq numsubp;
        Uq maxlc;
        Uq numlc;
        Uq maxmess;
        Uq nummess;
        Uq messlock;
        Sw cash;
        Uw lockword1;
        Uw lockword2;
        Uw icekey;
        Sw icecount;
        Uw *direcp;
        Uw dcleartime;
        Uw runtime;
        Uw wakeup;
        Uw type;
        Uw cause;
        Uq *messtab;
        Uq *subptab;
        Uq *tlctab;
        Uw tlcfree;
        Uq *dmaptab;
        Uq *fizadd;
        Uw faultaddress;
        Capl heir;
        Scratch *scr;
        } Prochd;
```

Figure 6.6: The **Prochd** Data Structure

deemed idle (**PROCSTATEIDLE**). For a process in **PROCSTATEKERNEL**, the **action** field contains a value indicating the system-call currently being executed by the process. The **stage** field indicates the stage to which a system-call action has been completed. The **currsubp** and **numsubp** fields respectively indicate the current subprocess and the number of defined subprocesses of a process. The **numlc** field holds the number of capabilities currently loaded. The number of empty mail boxes is stored in **nummess**. The **messlock** field acts as a semaphore for the process's mail boxes. The money used to pay for system-calls and transfers of money to and from objects is stored in the **cash** field. The process's two lock words are held in **lockword1** and **lockword2**. The *protected freeze* and *protected thaw* operations use the **icekey** and **icecount** fields. The last time the process's top level page table was cleared is stored in **dcleartime**. The **runtime** field holds the time when the process was last run. A process does not run until after the **wakeup** time has passed. The **type** field is a duplicate of the object's type field. The **cause** field is used for diagnostics; it identifies the reason for entering the scheduler. The **faultaddress** field contains the logical address of the memory access which resulted in a page fault. The **fizadd** field is a pointer to this **Prochd** data structure using a physical address. The **heir** filed contains the capability of an object to which a process's cash should be sent upon its demise. The **scr** field is a pointer, using physical addressing, to the **Scratch** data structure of the kernel executing the process. The **direcp**, **messtab**, **subptab**, **tlctab**, and **dmaptab** fields respectively point to a process's top level page table, *message table*, *subprocess table*, *Table of Loaded Capabilities (TLC)*, and the map of its address space. The **tlcfree** field contains the index of the head of the free list of the process's *TLC*.

The sizes of the *TLC*, the *message table*, and the *subprocess table* are specified when a process is created. These values cannot be varied during the life of a process. The **maxlc** field specifies the maximum number of loaded capabilities. The number of mail boxes - the size of the *message table* - in a process is specified by the **maxmess** field. The **maxsubp** field holds the maximum number of subprocesses.

Each process has a *Table of Loaded Capabilities (TLC)*. The *TLC* lists all the capabilities loaded by the process, the rights held by those capabilities when the

```
typedef struct Tlcentst {
        Uw vol;
        Uw serial;
        Uw pass1;
        Uw pass2;
        Uw srights;
        Uw urights;
        Uw base;
        Uw size;
        Sw displ;
        } Tlcent;
```

Figure 6.7: The **Tlcent** Data Structure

capabilities are loaded and the location of the loaded section of the capability in the process's address space. The index of an entry in the table is known as the *capability index* or *cindex* of a capability. The index can be used to identify a loaded capability. Empty entries in the table are formed into a doubly chained linked list. The **Tlcent** data structure is illustrated in figure 6.7. The **vol**, **serial**, **pass1**, **pass2**, **srights** and **urights** fields hold the values present when the capability is loaded. The **base** and **size** fields hold the offset from the start of the visible part of the object and the size of the visible part. Both quantities are in characters. The **displ** field holds the displacement between the beginning of the window holding the capability and the beginning of the address space.

```
typedef struct Subprocentst {
        Uw wakeup;
        Sw pcnt;
        sysstate regset;
        coprocstate coproc;
        Uw trap;
        Uq state;
        Uq priority;
        Uq pad2;
        Uq pad3;
} Subprocent;
```

Figure 6.8: The **Subprocent** Data Structure

The *Subprocess Table* holds the state of each of the subprocesses of a process. The state of the supervisor of a subprocess is stored in slot zero and the state of the

subprocess corresponding to the master capability of the process is stored in slot one. The *subprocess table* is constructed from **Subprocent** data structures (see figure 6.8). The subprocess is not scheduled to run until after the time in the **wakeup** field has passed. The **pcnt** field contains a pseudo program counter value used by *drive* processes. The state of the processors when the user process is pre-empted is stored in the fields **regset** and **coproc**. The number of the subprocess assigned to handle traps in the current subprocess is held in **trap**. The **state** field. Subprocesses may be non-existent (**0**), alive (**SUBPNORMAL**), or (**SUBPDEAD**). Subprocesses are allocated scheduling priorities to help select between subprocesses when a process has been scheduled to run. The priority of a subprocess is held in **priority**. The larger the value of the **priority** field the higher the priority of the subprocess. The value `0xff` is reserved for the supervisor.

A process's *address map* is stored at `0x100f000` in the logical address space, and the page containing it is marked read-only. The *address map* is a table of *capability index* values for memory locations. Figure 9.4 in chapter 9 contains sample code using the *address map*.

```
typedef struct Messentst {
        Uq chars;
        Uq subproc;
        Uq reserve;
        Uq matchlen;
        Sw money;
        Uw body [WORDSPERMESSBODY];
        } Messent;
```

Figure 6.9: The **Messent** Data Structure

The *message table* is built from **Messent** data structures (see figure 6.9). Each *mail box* - **Messent** data structure - can hold a single message. Empty mail boxes have `0xff` in the **chars** field, otherwise the **chars** field contains the length of the message. The **subproc** field holds the number of the subprocess to which the message should be delivered. The **money** field holds the amount of money sent with the message. Negative amounts of money cannot be sent by user processes. The **body** array holds the message sent.

A *mail box* may be reserved for the use of a particular subprocess by setting the

reserve field to the subprocess number. The value 0xff indicates that the *mail box* may be used for a message to any subprocess. A *match string* can be specified for a *mail box*. The *match string* is stored in the **body** of the message and the length of the string is stored in the **matchlen** field. Only messages which are prefixed by the *match string* can be stored in the *mail box*.

Each process has a *parameter page*. The *parameter page* is made up of the *parameter block* (see figure 5.7 in chapter 5) and the *message area*. The kernel reads fields from the *parameter page* when a system-call is made and returns values in fields of the **Param** data structure when the system-call returns. The fields and usage of the *parameter page* are described in chapter 5.

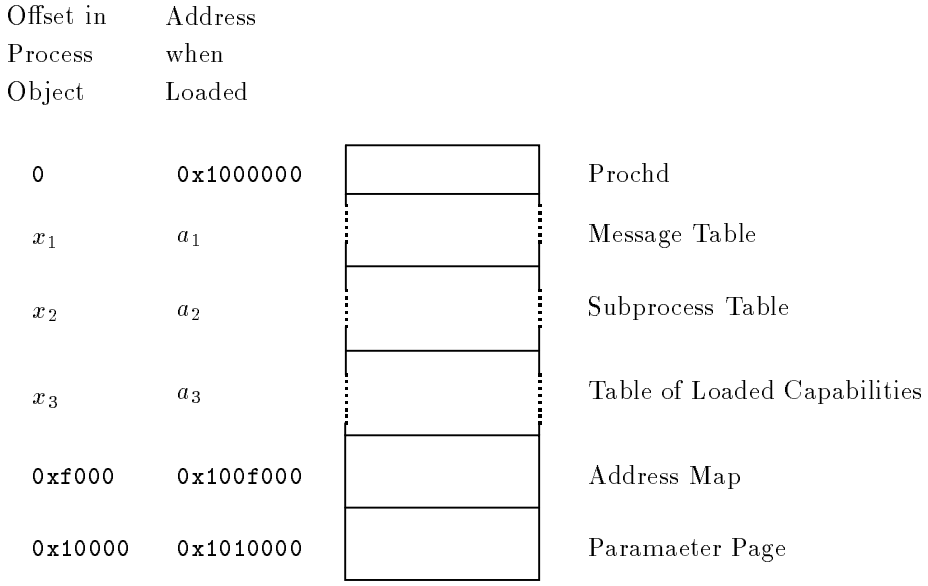


Figure 6.10: Layout of the Process Object

The layout of the *process object* is illustrated in figure 6.10. The **Prochd** data structure is concatenated with the *message table*, the *table of loaded capabilities* and the *subprocess table* to construct the first set of pages in the *process object*. The subsequent two pages of the object hold the *address map* and the *parameter page*. A significant feature of the *process object* is that the *message table* is entirely contained within the first page of the object to avoid extra page faults when sending a message.

6.2.4 Kernel Data Structures

Scratch data structures hold the state of each instance of the kernel while the kernel is performing an operation (see figure 6.11). The **error** field is used to hold an error code. Negative error codes indicate that a function cannot be performed at this time - the kernel, typically, retries the operation at a later time. Positive error codes indicate that a function cannot be completed. Error codes greater than 20000000 indicate that the kernel is in an inconsistent state. The numeric value of an error code indicates the routine in which the error occurred and the error that occurred: The rightmost 2 decimal digits indicate the error, and the more significant three digits indicate the routine.

Only some of the fields contain valid information. The set of valid fields is determined by the operation being carried out. The **vol**, **serial**, **pass1**, **pass2**, **srights**, **urights**, **base**, **limit**, **money**, **type**, **maxoff**, **maxsz**, **maxcap**, **offset**, **subpn**, and **cindex** fields hold information about the process or object being operated on. The **ventry** field holds an index into the volume table. The **head** field holds a pointer into the AOT for an object header. The **xhead** field is a pointer to another object header. The **capent** field points to a capability entry in the header of an object in the AOT. The maximum offset into an object is held in **objlim**. The size of the object header is held in **dopesz**. If the hash chain for an object header in the AOT is locked, **aotres** holds a pointer to the semaphore. The fields **pnum** - page number, **dirent** - second level page table entry, and **pte** - private page table entry are used for manipulating page table entries. Resolved physical addresses are stored in **fizadd**. The **darg** field contains a counter which is decremented while operations are carried out. It is set to a positive value whenever a kernel operation started or restarted. When **darg** is zero or negative, the operation is abandoned until it is retried. This use of **darg** ensures that no kernel action lasts for more than a certain time, chosen to be less than a scheduling time-slice. The field **prochd** points to the process header of the process currently being handled by the kernel and **param** points to the process's parameter page. The **kerneltick** field holds the value of the *tick counter*[6] when the kernel function was started.

[6]Each time a timer interrupt occurs the *tick counter* is advanced.

```
typedef struct Scratchst {
        Sw error;
        Uw vol;
        Uw serial;
        Uw pass1;
        Uw pass2;
        Uw srights;
        Uw urights;
        Sw base;
        Sw limit;
        Sw money;
        Uw type;
        Sw maxoff;
        Sw maxsz;
        Sw maxcap;
        Sw offset;
        Sw subpn;
        Sw cindex;
        Sw ventry;
        Header *head;
        Header *xhead;
        Captabent *capent;
        Sw objlim;
        Sw dopesz;
        Uq *aotres;
        Uw pnum;
        Uq *fizadd;
        Uw dirent;
        Uw pte;
        Sw darg;
        Prochd *prochd;
        Uw *param;
        Uq kerneltick;
        Uq dum1;
        Uq dum2;
        Uq dum3;
        } Scratch;
```

Figure 6.11: The **Scratch** Data Structure

Two arrays of integers are used in *process scheduling*. The arrays **mixvol** and **mixserial** hold the volume and serial numbers of processes to be scheduled by the kernel. Process's which are scheduled to wake up at time **FOREVER**[7] are not present in the *mix*.

6.3 Active Elements

The operations of the active elements of the kernel may be collected into five groups. *Object Memory Management* is concerned with the moving of parts of objects into and out of main memory, and with ensuring the stability of data stored in an object. *Capability Management* consists of providing all the operations on capabilities and ensuring that the set of operations is consistent. *Process Memory Management* governs the loading of views into a process's address space. *Message Management* is responsible for delivering and receiving messages. Finally, the *Process Scheduler* controls the scheduling of processes and subprocesses. Figure 6.12 shows the major relationships between these groups of functions and the data structures of the kernel.

6.3.1 Object Memory Management

Three elements of the kernel are connected with *Object Memory Management*: **scavenge**, **aotscavenge**, and a collection of routines that provide access to parts of objects. The routines **scavenge** and **aotscavenge** each have their own **Scratch** data structures and are periodically scheduled. These routines are responsible for removing entries from memory and the active object table. A number of routines are used to load and access a page of an object. The **refer** routine is central to accessing pages.

The Walnut Kernel's *Object Memory Management* is founded on the following guarantees:

A page remains in memory as long as any page table entry (including private page tables) contains a *valid* reference to it.

A page table remains in memory as long as any process's first level page table

[7] The end of time or **FOREVER** has the value `0xffffffff`

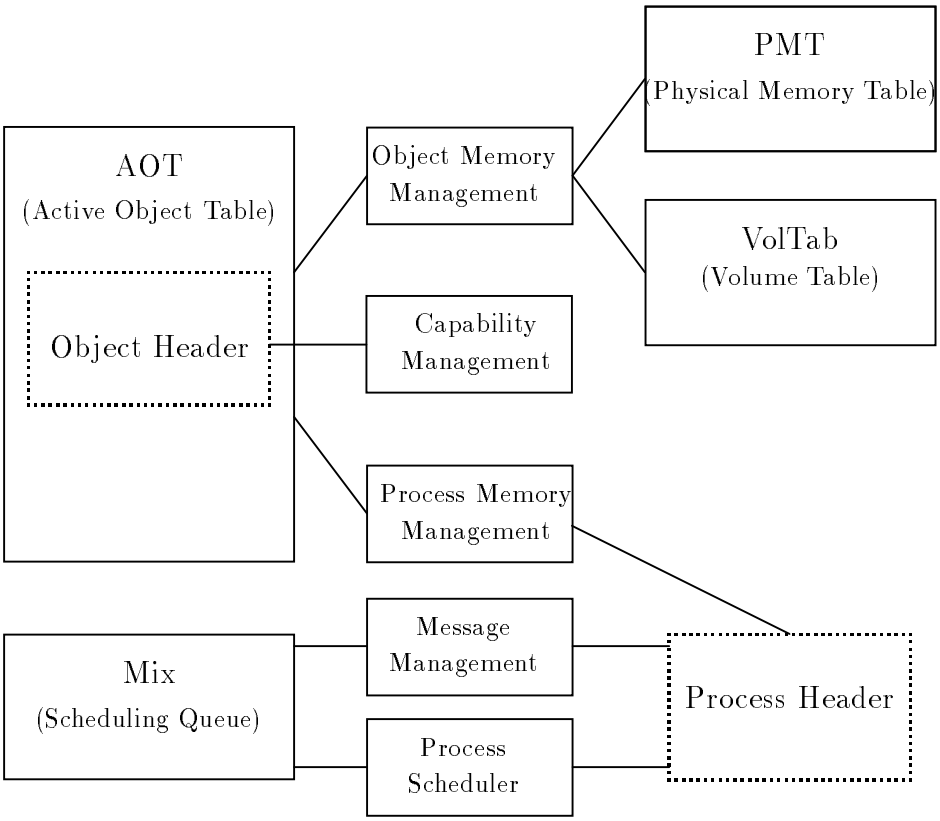


Figure 6.12: The usage of data structures by kernel functions

contains a *valid* reference to it, and as long as any page to which it refers remains in memory.

An object's dope remains in memory (in the AOT) as long as any of the object's page tables remain in memory.

The **scavenge** routine examines each entry in the *PMT*. The routine both copies dirty pages to disk and removes pages from memory. **Scavenge** stores the time at which the last pass through memory was completed and the number of the page currently being examined in its private **Scratch** data structure. The **scavenge** routine aims to examine every entry in the *PMT* in **DIRECTORYDELAY** seconds. When **scavenge** is scheduled it selects a target page frame number and attempts to examine each entry in the *PMT* between the last entry examined and the target entry. The target frame number is proportional to the number of seconds elapsed since a sweep of the table was completed divided by the **DIRECTORYDELAY**. To avoid flooding the *disk queue*, **scavenge** will exit if there are more than **SCAVWRITELIMIT** disk write operations outstanding.

When scheduled by the *kernel scheduler* the **scavenge** routine checks the state of any outstanding disk write operations. The state of each enqueued disk-write is stored in the **scavnote** hash table. There are four possible states for each entry: page accepted for write and selected for removal; page written and selected for removal; page accepted for write; and page written. For each completed write operation **scavenge**:

- If the page contains a page table and the page is not scheduled for removal - clears *PMT* **dirty** field.
- If the page does not contain a page table and the page is not scheduled for removal - clears the dirty bit in the page table entry for the page.
- If the page contains a page table and the page is scheduled for removal - the *object header* in the *AOT* is retrieved, the number of the disk block for the page table is written into the dope, **restabs** is decremented, and the memory page is released into the free list.
- If the page does not contain a page table and the page is scheduled for removal

- the number of the disk block for the page is written into the page table for the object, **respages** is decremented, and the memory page is released into the free list.

After processing the contents of **scavnote**, the routine attempts to schedule pages to be copied to disk and/or removed from memory. Pages are scheduled to be copied to disk if they are dirty. Only if less than a quarter of the memory is free are the data pages of an object scheduled for removal. An LRU (Least Recently Used) discipline is used to determine which pages should be removed. Scavenge clears the *present* bit in the second level page table entry on clean pages to ensure that a page fault occurs. A page fault is used to determine that a page has been accessed eliminating the need for a *use* bit in the page tables provided by hardware. The bottom four bits of the **state** field of the **fizent** data structure are used to hold a counter which indicates when a page was last accessed. The counter is incremented each time a page is examined by **scavenge** and cleared by any page fault on the page. The value of the threshold at which a page is determined to be old enough to be removed is determined by the demand for pages. The greater the demand for pages, the lower the threshold. Clean pages are removed by clearing the page table entry and adding the page to the free list. Entries are made in **scavnote** to indicate whether a page is being copied to disk or copied and removed. Second level page tables are removed only when there are no pages pointed to by the table resident in memory. Top level page tables are discarded and their pages released after **DIRECTORYDELAY** seconds has elapsed.

In addition to assisting in the removal of pages **scavenge** will retry failed attempts at loading a block from disk.

The **aotscavenge** routine performs a similar task to **scavenge**; however, it acts on the *Active Object Table*. The routine examines each entry in the *AOT* by stepping through the **aothash** table, and pursuing each hash chain. If either the **darg** of **aotscavenge**'s **scratch** is zero, or the routine has processed more than the fraction of the hash table's entries than the time since the last sweep through the *AOT* divided by **DIRECTORYDELAY**, then the routine will exit. It will resume work from where it left off when next invoked. Each **Header** is examined and the action

of saving the object header to disk is initiated if the time since it was last referenced is greater than a threshold value. The threshold is guaranteed to be greater than **DIRECTORYDELAY**.

An attempt by a process to reference an object can fail at many points. The actions taken by the kernel depend on the point of failure. In general, the kernel takes corrective action when a failure occurs and allows the reference to be retried when the process is rescheduled. References to an object are typically generated by processes. The majority of references are handled by the page translation hardware using the page tables set up by the kernel. A page fault occurs when a page is either *not-present*, as shown by absence of the page table entry *valid* bit, or an attempt is being made to write to a page which has been marked read-only. In either case, the **fixfault** routine is invoked by the fault handler. The fault handler stores the fault address in the **faultaddress** of **prochd**. On entry, **fixfault** examines the top level page table associated with the process and recovers the top level page table entry. If the top level page table entry is valid, the second level page table entry is recovered. If the fault has been caused by a *not-present* mark in the second level page table entry, then the **state** field of **fizent** is modified to indicate that the page has been accessed; the **present** bit in the page table entry is set; and **fixfault** returns to the process. If the fault was caused by an attempt to write to a page without *write* permission, an error is returned. Otherwise, the process's *address map* is used to find the *capability index* of the the capability corresponding to the **faultaddress**. The entry in the process's *Table of Loaded Capabilities* is copied into **scratch** and the **refer** routine is invoked. **Refer** checks the type of access being performed is valid and within the loaded section of the capability. The **vol** and **serial** fields are used to access the *Active Object Table*. If there is no entry for the object in the *bitmap* corresponding to the volume, an error is returned indicating the object no longer exists. If there is no entry in the *AOT*, the recovery of the **Header** for the object from disk is started, and **fixfault** returns. The *capability table* in the *object header* is checked to ensure that the capability loaded in the process's *TLC* is still valid. If the passwords do not match, an error is returned. If a page table for the object is already in memory, **refer** recovers the address of the page table from the

list of page tables in the *object header* and returns both a top level page table entry and a second level page table entry for the required access to **fixfault**. If a page table is not in memory, **refer** starts an operation to retrieve the page table from disk and returns to **fixfault**. If **refer** was successful in constructing the page table words, **fixfault** uses the words to complete the page tables and returns to the user program.

6.3.2 Capability Management

Capability management consists of three elements: the derivation of capabilities, the restriction of capabilities, and the revocation of capabilities. The majority of these operations are conducted solely in the *header* of the object the capabilities relate to.

The derivation of capabilities is performed by the **addcap** routine. The **vol**, **serial**, **pass1** and **pass2** of the capability to be derived from is passed to **addcap** in the **scratch** data structure. The rights mask and the parameters for the coverage of the derived capability are passed in the fields **srights**, **urights**, **base** and **limit**. Before a derivation can be performed, the rights of the presented capability are checked to ensure that it has **SRDERIVE** right. If no *derive* right is present, an error is returned. If the **SRMULTILOAD** right is absent, the password 2 of any derived capability is coerced to the value of the password 2 of the presented capability. With the exception of the *suicide* right and the *message* rights of processes, derivation consists of returning the logical-and of the rights of the nominated capability and the rights mask supplied by the caller. The memory area covered by the capability is determined by the intersection of the region covered by the presented capability, and the area covered by adding the **base** to the start of the region covered by the presented capability up to the extent provided by the **limit**. The **SRSUICIDE** right can be added to the children of any capability.

The *system rights* field for a process is treated differently from the *system rights* of a data object. The last eight bits of the **srights** field limit the subprocesses to which a message may be sent by using this capability. The bits may contain `0xff` for all subprocess, or `0xfe` for all subprocess other than subprocess zero, or a subprocess

number. Only more specific or equal subprocess destinations for messages can be derived. Furthermore, it is required that the capability base be zero for a derived capability which has **SRSEND** right.

The revocation of a capability is a two phase process. The **delcap** routine deletes a capability and all the derivatives of the capability from the *capability table* in the object's *header*. References to a deleted capability loaded in the address space of a process remain valid until a page fault results from attempting to access the deleted capability. The top level page table of every process is guaranteed to be discarded and replaced with an empty page table every **DIRECTORYDELAY** seconds. The first access after the top level page table is replaced results in a page fault, and the deletion of the capability is noted. The Walnut Kernel guarantees that a capability is unavailable to all processes within **DIRECTORYDELAY** seconds of revocation.

6.3.3 Process Memory Management

Process Memory Management consists of two operations: mapping views into the address space of a process and removing a view from the address space of a process.

A process requests the loading of a view into its address space using the **LOADCAP** system-call. When presented with the capability to be loaded, the Walnut Kernel attempts to locate the *object header* in the *AOT*. If the object is not represented in the *AOT*, recovery of the *object header* from disk is initiated. When the *object header* is present in the *AOT*, the appropriate entry in the *capability table* is recovered. If the recovered capability lacks **SRMULTILOAD** right and password 2 of the capability is not equal to the serial number of the process, then an error is returned. Otherwise, the values representing the area covered by adding the **base** to the start of the region covered by the capability to be loaded up to the extent provided by the **limit** are placed in the **base** and **limit** fields of *TLC* entry. The *rights* fields of the **Tlcent** data structure are filled in, and the system-call returns successfully. The capability's **cindex** is written into all *address map* entries covered by the view.

The removal of view from a process's address space is trivial. The fields in the

process's *address map* corresponding to the memory address range occupied by the mapping are zeroed, and the entry in the process's *TLC* is invalidated and linked into the list of free *TLC* entries.

6.3.4 Message Management

The message mechanism enables both the transfer of information between processes and control of the scheduling of processes. When a process is created an array of **Messent** data structures is allocated within the process object. Only the kernel can access mail boxes directly.

A message is sent by the kernel copying a set of bytes from the source process's message area (the remainder of the page in a process's address space that contains the parameter block) to the destination process's mail box. For this to occur, it is necessary for the capability used to identify the destination to start at offset zero from the beginning of the process, and to have the **SRSEND** right. The recipient process may be explicitly named using the *external* form of the send system-call; alternatively it can be implicitly named by sending a message to a capability loaded into the sending process's address space using the *internal* form of the send system-call. The latter method is more efficient as it eliminates the need for the kernel to verify the access rights of the destination process object. It reduces the send operation to a memory copy. The **transmit** routine implements both the *internal* and *external* forms of message sending.

The **transmit** routine transfers money from the sender's **cash** word to the receiver's **cash** word. The transfer is made after the message has been placed in the mail box.

Messages are always directed to a subprocess. When a message is sent to a subprocess, the wakeup time of the process is set to the current time, and the process is placed in the *mix*. Whenever a mailbox contains a message, a subprocess cannot change its wakeup time from the current time. Hence non-frozen processes with pending messages are always scheduled to run.

The **receive** routine is called by the *receive* system-call. The routine can take a match string as a parameter which allows a subprocess to retrieve messages begin-

ning with the specified string. If no match string is provided, then the message in the first mail box containing a message for the current subprocess will be retrieved.

When a process is created all of its mailboxes are *closed*, except for a single mailbox allocated to subprocess zero. After the process has completed its initialisation, the process can *open* its mailboxes setting the fields which reserve mailboxes for messages with specified prefixes and specific subprocesses. Messages sent before the process has opened its mailboxes are not delivered. A system-call allows mailboxes to be closed.

6.3.5 Process Scheduler

At present a round robin scheduler is employed within the kernel to schedule all processes with a specified wakeup time. Processes which have selected to wait forever are removed from the scheduling queue. The scheduling queue within the kernel is known as the *mix*.

When the *process scheduler* - **macro_schd** - selects an element from the *mix*, the volume and serial number of a process are passed to the **startproc** routine using the **scratch** data structure.

Startproc must ensure that the set of pages critical to the operation of a process are loaded into memory before attempting to transfer control to the process. The set of critical pages consists of two parts: the *dope* for the object holding the process, and the set of *necessary pages* of the process. The *necessary pages* of an object are the pages containing the **Prochd**, the *Table of Loaded Capabilities*, the *subprocess table*, the *message table*, and the *parameter page*. The *AOT* entry for the process is checked. If the entry is not present, retrieval of the dope is initiated. When the *AOT* entry is recovered, the top bit of the **type** is tested to ensure that the entry object contains a process. If the object does not contain a process, an error is returned.

A lock is placed on the message table by locking **messlock**. If the lock operation fails, a message is currently being sent to this process, and **startproc** exits with a negative error. If the lock operation succeeds, **startproc** guarantees to clear the lock before exiting.

The **PROCSTATEMIX** bit of the **state** field of the *process header* is set.

Prochd's **wakeup** field is examined next. If the process wakeup time is **NEVER** (also known as **FOREVER**), **startproc** exits with a positive error which informs the scheduler to remove the process from the *mix*. If it is too early to wakeup the process, a negative error is returned.

The time the process's directory (top level page table) was last cleared is checked to ensure that the top level page table is still valid. If **scavenge** has had an opportunity to remove the page, a new page is created. This has a side effect of ensuring that all capabilities are revalidated when they are next used.

Once the top level page table has been established, references to the *necessary pages* page table are inserted in the top level page table. If the *necessary pages* are not present in memory, they are recovered from disk.

The process state is checked. If a process is dead (**PROCSTATEDEAD**), an error is returned, and it is removed from the *mix*. If a process is in probate (**PROCSTATEPROBATE**), the process attempts to send a message to its **heir** containing the process's **cash** and the process's name. Only 255 attempts are made at delivering the message to the **heir**. If the **heir** is unable to accept the message, the process changes to **PROCSTATEDEAD** and the cash is lost.

A check for new messages is performed, and messages for non-existent subprocesses are erased. When a subprocess receives a message it is marked runnable. Messages for subprocess zero are processed and the operations performed.

If the process is either frozen (**PROCSTATEFROZEN**) or subject to a protected freeze (**PROCSTATEPFROZEN**), **startproc** exits allowing other processes to be scheduled.

At this point, **startproc** chooses the subprocess to execute. If the **reserve** field of the process's **Param** data structure (*Parameter Block*) is non-zero, the subprocess that was executing at the end of the last time-slice is re-started. Otherwise, the subprocess with the highest priority and with a wakeup time less than the current time is selected. If no runnable subprocesses are found, **startproc** returns with a negative error.

Having selected the subprocess to be run and ensuring that the pages critical to the operation of the process are in memory, **startproc** returns to the **macro_schd**

routine.

If an error was returned by **startproc**, **macro_schd** selects a new subprocess from the *mix* and calls **startproc**. Only when a runnable process is selected, is the remainder of **macro_schd** executed.

The remainder of **macro_schd** is devoted to executing the selected process. The actions of **macro_schd** fall into three categories:

- If the process is currently handling a page fault (**PROCSTATERFAULT** or **PROCSTATEWFAULT**), the **fixfault** routine is called. If the fault is fixed, the process state is set to **PROCSTATENORMAL**, the subprocess state is restored, and the process is executed. If the fault cannot be fixed immediately, the process is returned to the *mix*.

- If the process is currently performing a system-call **PROCSTATEKERNEL**, the **kcact** routine is called to handle the system-call. If the system-call has been completed, **post_kcact** is invoked to clear any fields in **Param** of data which are not to be returned to the process, the subprocess state is restored, and the process is executed. The process is returned to the *mix* if the system-call cannot be completed.

- If the process is in **PROCSTATENORMAL**, the subprocess state is restored, and the process is executed.

6.4 Persistent Elements

Figure 6.13 illustrates two typical layouts for a disk containing a Walnut Kernel volume. The layout of disk is fairly flexible as the only elements which have fixed locations are the *Disk-ID-Blocks*. The first *Disk-ID-block* is located in the first block of a *data volume* or at a location defined by a compile time constant in the kernel for a *bootable volume*. The duplicate *Disk-ID-Block* is always located in the last block on the disk. The *Disk-ID-Blocks* and the Bitmap contain information relating to the whole disk. The remainder of the disk is used to store objects.

The *reserved area* is located at the front of the disk and is typically used to hold

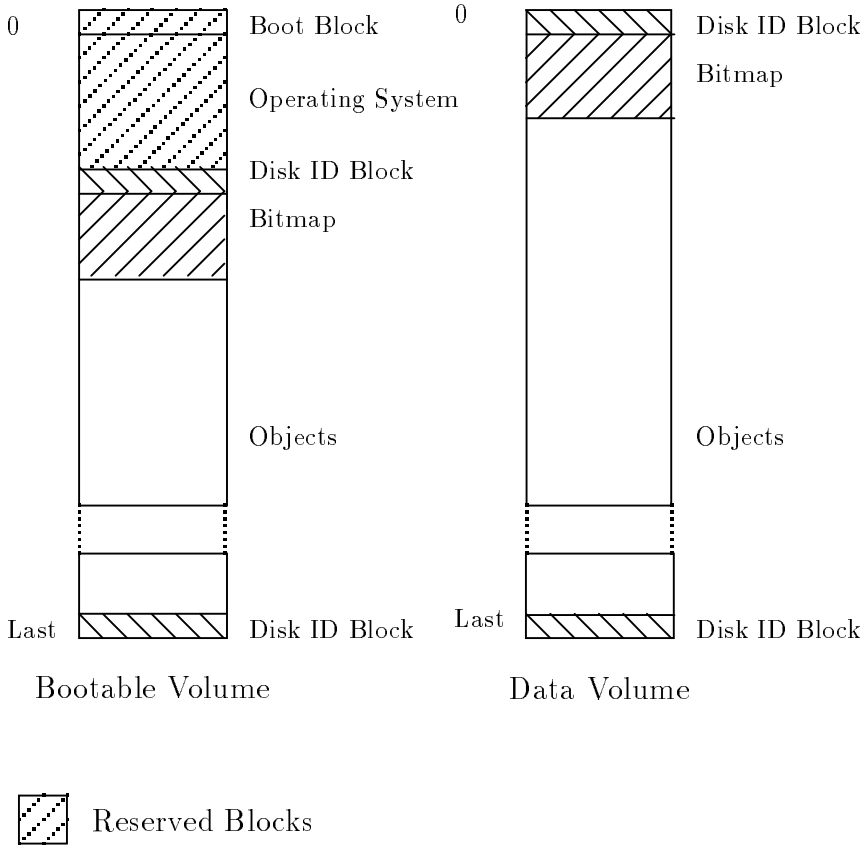


Figure 6.13: Disk Layout

the operating system and the boot block. The reserved area is ignored by the kernel. Typically *data volumes* do not have a reserved area.

The *Disk-ID-Blocks* are duplicated to assist in the recovery of information from a corrupted volume. The location of the *bitmap* and the identification of the *reserved area* simplify the recovery procedure. Furthermore, the duplicate *Disk-ID-Blocks* allows a volume with a damaged *Disk-ID-Block* to be mounted.

Unlike the Password-Capability System and file based operating systems, the Walnut Kernel does not possess a File Allocation Table (FAT) or its equivalent. A FAT is a lookup table that is used as the first step of translating a name into a disk location. Instead the Walnut Kernel uses part of the objects name - the low order bits of the object's serial number - to locate an object.

A file allocation table allows names to be independent from position. The absence of a FAT structure initially appears to be a critical problem for the Walnut Kernel as it ties objects to particular locations on disks. The failure of a disk block containing the header of an object would result in the loss of the object. In practice the presence of a FAT has little advantage for the operation of the Walnut Kernel. Three issues are significant: reliability, reconstruction and security.

Reliability and reconstruction are related issues. Enhancing the reliability of a system's data storage reduces the likelihood of needing to reconstruct lost information. Both processes require redundancy in the data to infer the lost components.

A file allocation table provides a centralised listing of all the items on a disk. They are frequently duplicated to several locations on a disk to prevent the potentially catastrophic results of loss of the table - the loss of all translation information and hence the loss of access to all the objects on the disk. The entries in a FAT either contain information relating to an object or point to block of information about an object.

Redundancy has been built into the Walnut Kernel's persistent representation. The Walnut Kernel has a pair of *Disk-ID-Blocks* which are used to identify the volume and, the header pages of an object are recognizable by signature words at both the top and bottom of the page allowing pages to be identified. The signatures can be used in reconstructing damaged bitmaps.

The presence of a FAT would not significantly enhance the reliability of the system above the level provided by the current scheme as:

- Loss of a header block results in the loss of all derived capabilities for an object. Although a FAT could contain a duplicate of all the information contained in the header of an object the significant penalty in terms of both space and speed would be unacceptable.

Nor would reconstruction be significantly enhanced by the presence of a FAT as:

- The data regarding the location of header pages is already duplicated in the bitmaps.

The use of the low order bits to identify the disk location of the header page of an object has a minimal effect on the security of the system (see section 10.3).

The absence of a FAT is significant in terms of restoring data from a backup media. Under the Walnut Kernel it is not possible to restore an object if the block on which the header page is required to reside is damaged. The disk storage mechanism lacks position independence. A system using a File Allocation Table would not be constrained to restoring an object to the same physical location. However, the majority of modern disks employ re-mapping techniques which translate accesses to failed sectors to replacement sectors seamlessly. Thus the hardware appears to be faultless on restore even if there has been a failure.

A more subtle problem is present. Under the Walnut Kernel it may not be possible to backup a single object and restore that object at a latter date. The problem stems from the lack of position independence. If a block required for the header page is already in use then it is not possible to restore the object.

The backup and restoration of capability based systems is an open question. Problems exist as to the meaning of partial backups and restores; the semantics of the deletion of an object is changed as the object may have been backed up. The security of backed up objects is also subject to significant doubts. Solutions employing cryptography to protect the contents and signatures to prevent tampering have been put forward. These mechanisms are vulnerable to cryptographic attacks with selected plain texts and hence require extremely strong algorithms to remain secure.

Furthermore as they rely on an encryption key derived from secret information held by the system it is typical for backups of objects on the system to be generated using keys related to keys used to backup other objects on the system. Relying on related keys decreases the system security as the compromise of the encryption of one object can give clues to compromising other objects.

The Walnut Kernel does not address the problems of backup - apart from the trivial case of sector by sector backup and restoration of the complete system - at this stage. Our view is that backing up an object creates a new object with a new name, which should be distinguished from the original, even if its body and passwords are the same.

6.5 Small Windows & Private Page Tables

Second level page tables are typically associated with objects. However, to provide small windows, a collection of second level page tables - known as *Private Page Tables (PPT)* - is associated with each process. In the description of the system, the second level page tables associated with processes were neglected for simplicity. This section describes the operation of small windows and *PPT*s.

A compile time constant determines the number of top level page table entries devoted to implementing small windows. These entries are separated from the page table entries used for large windows by the entry used to hold the second level page table for the process object (see figure 6.14).

Private Page Tables are subject to scavenging and require replacement if the table may have been removed. The operation of *PPT*s is analogous to the operation of top level page tables. When a reference is made to an object using a capability loaded into a small window and the *PPT* is invalid, the capability is validated, and a pointer to the required page of the object is made. The only significant differences between the usage of *PPT*s and top level page tables are that the *PPT*s are a per process data structure, and the dirty bits in the object's page tables need to be set when a write operation is performed on a small window. To ensure that the dirty bits of the object's page tables are correctly updated, entries in *PPT*s created by reads are not marked writable, even if the capability allows the object to be written

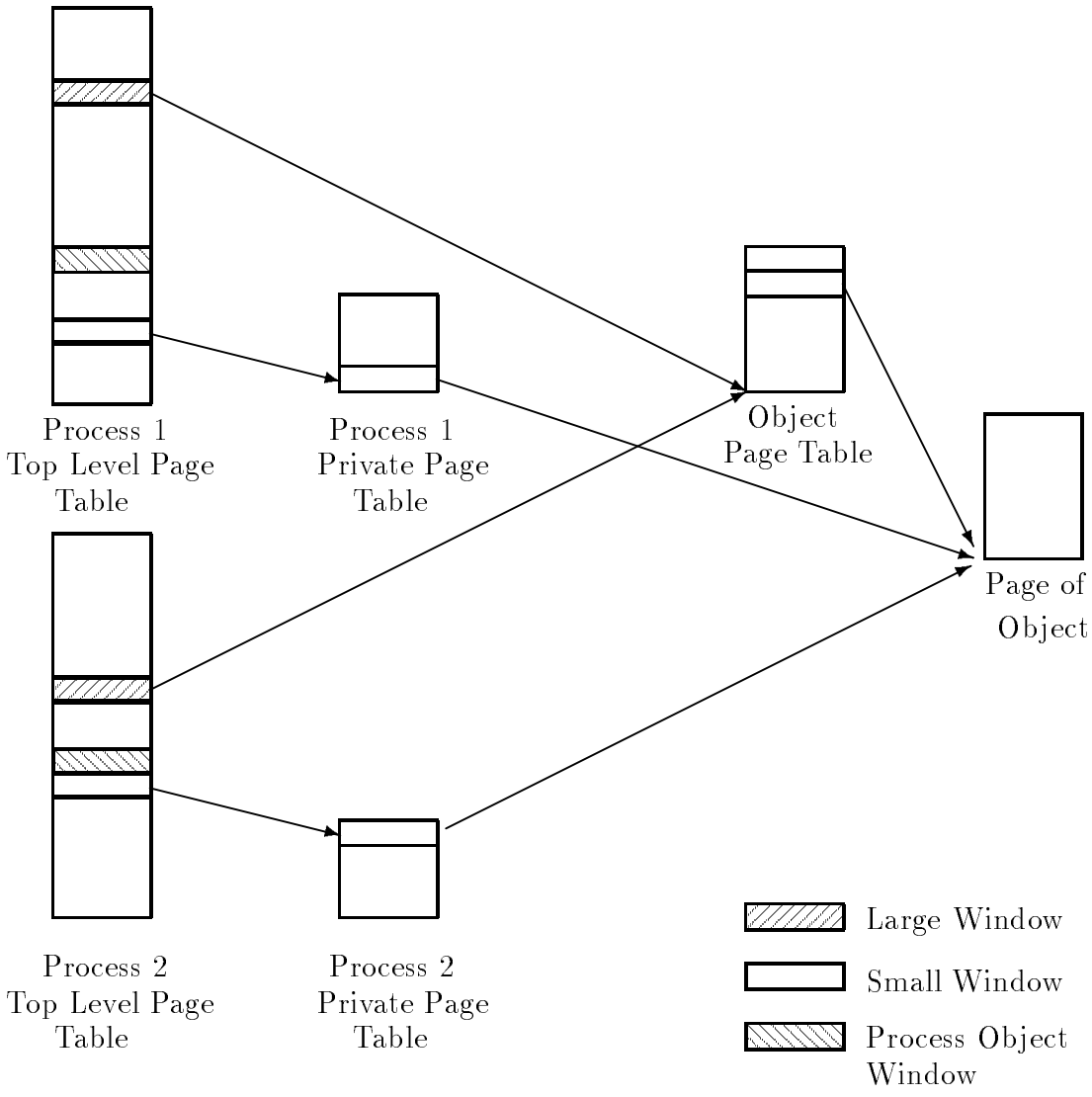


Figure 6.14: Windows and Objects

to. This causes the first write operation to page fault, and allows the dirty bit in the object's page table to be set. The *PPT* entry is then marked writable.

6.6 System Architecture

The design of the Walnut Kernel is divided into common and hardware architecture dependent components (see figure 6.15). This division in the design is reflected in the implementation which simplifies the task of porting the kernel to other processors and allows the identification of features required to allow the implementation of the Walnut Kernel on other architectures.

The components of the Walnut Kernel common to all implementations include the system-call interface, and memory and capability management. The low level functions that support the common components of the system are architecture dependent. The architecture dependent components include the low level device drivers and the kernel interface to essential system hardware.

The functions of the modules of the kernel are defined as follows:

System-Call Interface module as the interface between user programs and the Walnut Kernel. The module performs the initial verification of arguments to system-calls and ensures that only permitted information is returned to the user

Subprocess Zero Interface interprets messages sent to subprocess zero of a process. This mechanism augments the system-call interface.

High Level Functions have been described in section 6.3.

Kernel Scheduler module is driven by the timer interrupt and it calls device driver routines which require regular execution and the process scheduler to allow pre-emption of user processes

Device Drivers are not logically part of the kernel. In some implementations the collection of interrupt driven modules that interface with the hardware execute within the kernel. In other implementations hardware may perform

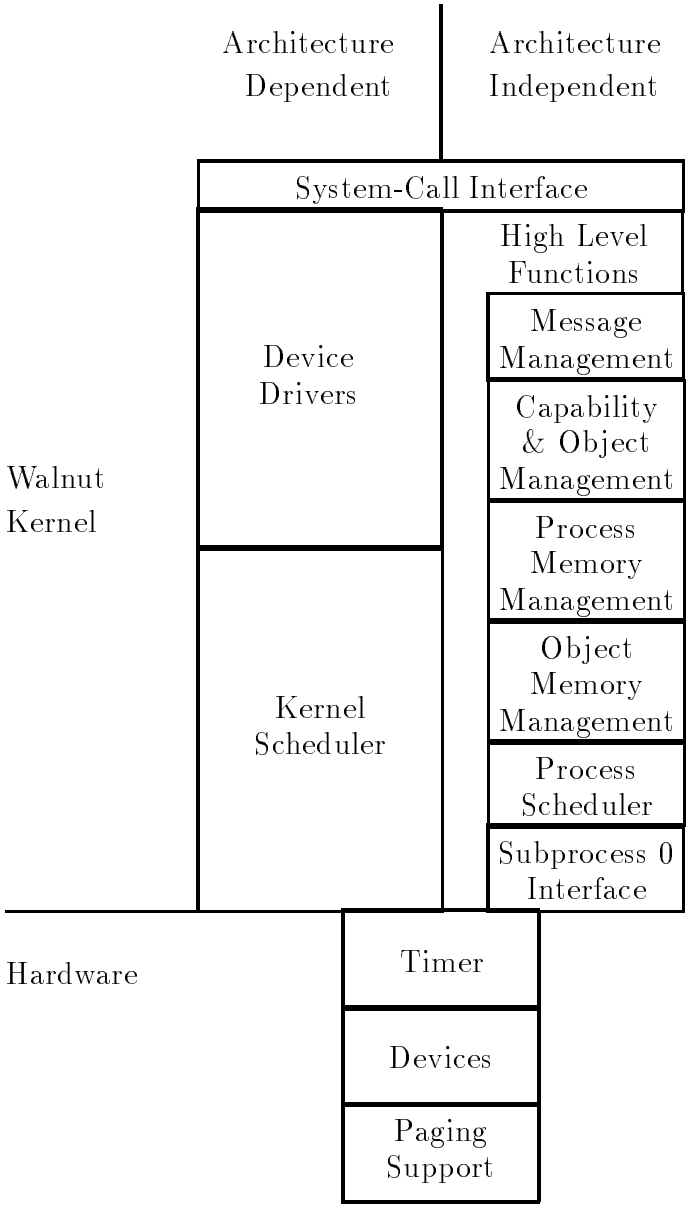


Figure 6.15: Components of the Walnut Kernel

all the functions required[Cat88]. In both cases a user level process gains access to the device through a number of pages of physical memory accessed by the capability mechanism.

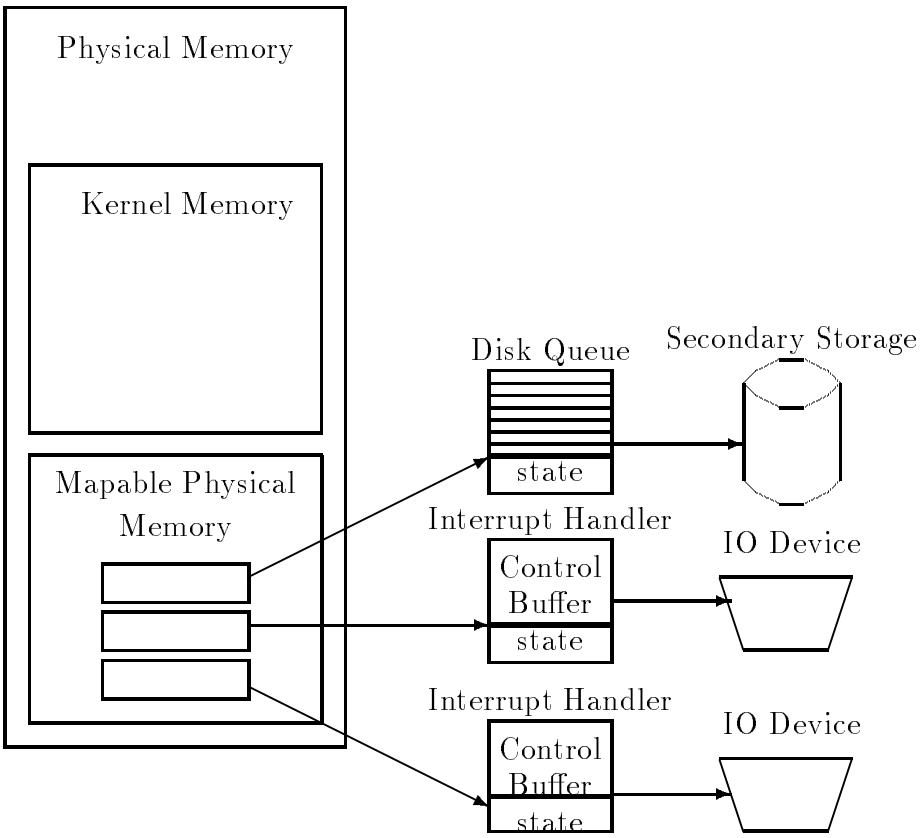


Figure 6.16: Organization of the Walnut Kernel

This design applies to both single processor systems and multiprocessor systems. Figure 6.16 presents a high level representation of the organization of the Walnut Kernel.

Interfaces to devices are placed into memory regions which can be mapped into ordinary processes. The remainder of the physical memory accessible to the processor is used for paging. The architecture exploits the sharing of memory to allow the kernel and user processes to communicate with devices. In systems with purpose built devices the device can interrogate the shared memory area directly to determine its actions. Devices which require a processor to provide close control of the stages of their operation are supported using low level device drivers which are

scheduled using the kernel scheduler and interrupts. The low level drivers transfer data and operate on instructions found in the shared buffer area.

The disk device / device driver is unique in that it is the only device which has direct access to the tables used by the kernel. The disk device notifies the kernel that a page has been brought into memory by setting a bit in the physical memory table indicating that the page is present.

The kernel is unaware of the operation of other classes of input / output devices. All other devices are handled by user level device drivers which interact with the devices using the standard capability mechanism to map in the memory shared with the physical device or the low level device driver.

6.6.1 System-Call Interface

The system-call interface is the mechanism through which user processes explicitly communicate with the kernel.

The *System-Call Interface* consists of both architecture dependent and architecture independent elements. The system dependent component provides a transition of privilege level to supervisor mode and starts kernel code at a fixed address. The portable component communicates directly with the majority of the high level modules of the kernel. The user process transfers information to the kernel by filling in a parameter block that is accessible to both the kernel and the user process and using the system dependent mechanism for switching to supervisor mode.

The *System-Call Interface* performs some checking on calls to ensure that requests are valid and legal before invoking the appropriate kernel routines to perform the requested operation. After completing the required kernel operation, the parameter block containing the values to be returned to the user process is post processed to erase fields which contain information not to be passed back to the user. This practice simplifies the task of showing that the kernel does not leak information to user processes.

The programmer's view of the parameter block, user level system-calls and message operations are discussed in detail in appendix A.

The 'reserve' field of the parameter block is used to select the type of system-

call to be executed. Furthermore, setting the 'reserve' field to a non-zero value guarantees that only the current subprocess of a process is scheduled until the field is set zero. It is imperative that subprocesses set the the reserve field to the system-call identifier before setting any of the other fields of the parameter block. Failure to do so may result in the parameters of a system-call being corrupted by the operation of another subprocess of the process. At the completion of the system-call, a process must copy any values required from the parameter page before clearing the 'reserve' field.

Other mechanisms were considered for transferring information between user processes and the kernel. These included using registers to pass parameters, passing a pointer to parameter block located in the processes address space, using a block of data located on the stack, and using multiple areas for both each subprocess and each type of system-call.

The use of registers for transferring data was rejected as it places constraints on the choice of processor and platform. This is not compatible with the design principles.

Using the stack to pass parameters was rejected because of the extra effort required to check the validity of the parameter area on each system-call to prevent attempts to read memory outside the area permitted. Furthermore, using the stack would have the potential to allow programmer's to create errors which would be difficult to trace. Many processors have stacks which grow towards zero allowing string operations which overflow the space allocated on the stack to corrupt the parameters of subroutines which have not yet completed. Routines constructing the parameters of system-call which use other subroutines are especially vulnerable as the effect of a system-call varies widely with minor changes to the parameter block.

The use of a parameter block fixed in a process's address space was motivated by the following considerations:

- Fixing the block in the address space eliminates the need to test whether the parameter block lies in an area of memory validly accessible to the process. Using fixed memory locations simplifies the task of ensuring that the kernel does not leak information and cannot be tricked into altering system state

invalidly.

- Including the parameter page in the set of pages required to be present in memory for a process to execute ensures that the parameter page is always present while a process is running, so processes can make system-calls without the risk of a page fault.

The current design supports passing a wide range of parameters to the kernel as the message area can be used to pass any data structure (slightly smaller than a page in size) required to the kernel. The design offers great flexibility and potential to extend the design.

We decided not to allow subprocesses of a process to execute in parallel. This eliminated the requirement to provide multiple parameter blocks. Should it be deemed desirable to allow subprocess to operate in parallel the kernel can be easily modified to allow subprocesses to operate in parallel until a subprocess sets the reserve field to a non-zero value. Other subprocesses would then not be scheduled until the reserve field is cleared. Mutual exclusion on the 'reserve' field prevents simultaneous system-calls from the same process simplifying the kernel design. In effect, system-calls would become 'critical regions' which only one subprocess could enter at a time.

6.6.2 Subprocess Zero

A special subprocess known as *subprocess zero* was introduced in the Walnut Kernel. Subprocess zero is an extension of the kernel that interprets specially formatted messages sent to a process. It allows a process to control the execution of another process by sending a message to the other process's subprocess zero. This mechanism supports only a limited number of operations on a process including suspending / resuming the running of the process, starting a subprocess and enquiring about a process's status. The mechanism is implemented by parsing any pending subprocess zero messages at the start of a process's time-slice and then performing the action associated with the message. If the content of the message is not recognised the message is ignored.

There are two alternative implementations of this mechanism: extending the system-call mechanism by adding calls to perform the tasks currently handled by subprocess zero, or requiring all processes to provide an executive subprocess which handles the messages and performs the actions currently delivered to subprocess zero. The current mechanism was selected because it provided the conceptual equivalent to the latter mechanism with the immutability of the former mechanism. The design considers the functions performed by subprocess zero to be semantically associated with a process, but requires that all the functions be present in each process in a consistent form.

6.6.3 Device Drivers

The UNIX model of a device driver consists of two halves both of which are built into the UNIX kernel. The top-half of the driver interacts with user processes. It runs in synchrony with the user process and may suspend itself using the *sleep* system-call. The bottom-half of the device driver runs asynchronously with respect to user processes. It is typically interrupt driven.

The Walnut Kernel handles devices in a manner which differs significantly from the UNIX model. Under the Walnut Kernel a user level process performs the tasks of the top-half of the UNIX device driver. The user process (also known as the device manager) communicates with either the hardware or a low level driver using a shared buffer area. If interrupts or the system I/O address space need to be addressed to operate a device then a low level driver is required. The low level device driver is compiled into the kernel, and typically performs the task of handling interrupts and moving data between buffer pages in the unity mapped region of the kernel and the device. Low level device drivers use only physical addresses to access the device, and the data and control buffers. Although the low level driver is compiled into the kernel it is logically not part of the kernel as it operates on only a defined area of physical memory in response to interrupt events.

In a multiprocessor system the low level driver could be replaced with specialised hardware writing directly into shared memory.

This method of implementation avoids a number of potential security holes:

- There is no requirement for a special class of user level processes that can access physical I/O. Only the kernel has direct access to hardware functions, eliminating a common avenue of attack.

- Careful coding of the low level device driver can insure that the kernel does not leak information or alter data which does not belong to the device.

This method also provides a number of significant performance advantages as it allows the low level drivers to respond rapidly to interrupts. Two factors contribute to the performance of this mechanism:

- The use of physical addresses and buffers statically allocated in memory: This avoids the overheads of paging and ensures the presence of buffers in memory.

- The direct use of the interrupt, which eliminates the need to invoke user routines upon an interrupt.

It should also be noted that the Walnut Kernel is at least as efficient as the UNIX model in terms of the number of memory copies required to transfer data from the device to the user program. Under UNIX at least 3 copies are required to transfer the data from the device to the user process[8]. The Walnut Kernel typically uses 3 copies, but requires only 2 copies. The two copy scenario is achieved by providing the capability of the memory buffer to the user level process that is going to use the data. In this scenario data is copied from the device into the buffer and then retrieved by the user process.

6.6.4 Kernel Scheduler

The Kernel Scheduler periodically invokes routines that have been registered with it. These routines include the Process Scheduler and device drivers. It indirectly activates the scavenger routine by calling the Process Scheduler. The Kernel Scheduler is driven by a hardware interrupt and invokes each registered routine in turn. The Kernel Scheduler provides the ticks which are used to pre-empt running tasks.

[8]device – bottom-half – top-half – user process

6.6.5 Discussion

The architecture enhances the portability of the kernel. The kernel interacts with only one class of input/output device - disk drives. The interface to this class of devices consists of a disk queue. By standardising the queue interface the kernel can be made portable up to the disk queue interface. Any system dependent code is placed on the hardware side of the disk queue.

The logical separation of devices from the kernel allows the easy introduction of special purpose processors on multiprocessor systems. This provides an opportunity to use a special purpose processor which handles disk transactions freeing the general purpose processors to handle user code. Other special purpose IO processors may also be directly mapped into the memory of a processor. This makes the kernel extremely flexible as new peripherals can be made available without modification to the kernel by mapping the devices into the address space and using user level programs to control the new devices.

This architecture is slightly less efficient at handling interactive programs than conventional systems such as UNIX. When a process blocks on IO in a conventional system, the process scheduler is notified that the process cannot be run, and the process is typically removed from the short term scheduling queue until the IO operation has completed. Under the Walnut Kernel this optimisation is not available to the processes which manage devices. Device drivers are separated from the kernel and this prevents the scheduler from being notified that a process is blocked on IO. The name of the device manager process could have been stored in the data accessible to the low level device driver and messages sent to the process on device activity at the cost of increasing the coupling of the device driver and the kernel, and additional system load resulting from messages sent by the kernel to the device manager. The absence of this optimization results in a small loss of efficiency. Careful programming practices, such as surrendering the processor quickly on determining IO is not possible, reduce the impact of this architectural limitation. Efficient blocking IO is provided for other processes through stream IO libraries. These libraries put the reading process to sleep when the process is blocking on input. When more data is available, the reading process is reawakened by a message from the writing process.

A significant feature of the kernel - to the user - was the decision to make the kernel occupy a region of the process address space. From a system design point of view, making the kernel permanently resident in memory was of greater importance. By making the kernel permanently resident in memory and a part of the user process address space, a number of significant problems were avoided and the task of implementation simplified. Development and debugging time was saved.

The elimination of page faults within the kernel simplifies the virtual memory system. In systems where paged kernels are employed, it is possible for the kernel to be stalled while paging in a critical component. In addition the problem of page faulting while handling a page fault has been eliminated, resulting in a reduced size of the kernel stack and reduced complexity in handling page faults. As page faults should not arise in supervisor mode, the leaking of kernel powers to user processes is prevented.

A unity mapping of the kernel memory region into the process address space allows the kernel to switch between virtual and physical addressing schemes easily. Unity mapping memory simplifies the design and implementation of low level device drivers.

A major advantage of placing the kernel at a fixed location in the physical address space is that it enables use of a logic analyzer to be used to trace the addresses of executing instructions. During the early phases of development the logic analyzer proved invaluable as it provided a mechanism for determining the cause of failures at the instruction level.

6.7 Design Issues

6.7.1 Pages versus Segments

The overriding concerns of portability forced a much coarser granularity of protection onto the Walnut Kernel than was present in the Password-Capability System. Although it would be desirable to have the 4 byte protection granularity found on the Password-Capability System available, this would require the use of either segment registers, where the processors have these available. Is not possible to provide

extremely fine levels of control where only the paging mechanism is available using memory access instructions. It would be possible to provide fine grain control on accesses to an object mediated by the kernel. This could be implemented as a system-call which would recover a set of bytes specified by the caller. This mechanism suffers from significant overheads and was deemed too inefficient to be widely used.

The Walnut Kernel has a significant advantage over the Password-Capability System in that the Walnut Kernel supports approximately 250 capabilities in a process address space at a time. This number could be easily increased should the number be considered insufficient. The older system supported only 32 capabilities at a time. Although it would have been possible to rewrite the Password-Capability System's kernel to handle greater numbers of concurrently loaded capabilities, the modifications would be far greater than those required on the Walnut Kernel.

The division of windows into small and large types reflects a compromise between the space efficiency provided by large windows and reduced granularity offered by small windows. Although the compromise adds complexity to the view of the system seen by the programmer, it allows programmers greater flexibility in dealing with, and control over, objects being manipulated by Walnut Kernel programs.

6.7.2 Multiple Processors

A peer implementation was selected for the design of the multiprocessor system rather than a master/slave implementation. The use of a uniform architecture for both uniprocessor and multiprocessor systems eliminates optimisations available for the two classes of implementation. However, it significantly eases the task of porting the system and reduces development costs. In the case of multiprocessor development it allows low cost compatible uniprocessor hardware to be used in the development phase before shifting to multiple processors for the production phase.

Supporting shared memory multiprocessors is relatively simple in that the kernel can access information available to other processors by uttering addresses within the shared component of the system memory. The code and data used by the kernel can be shared in total. The only exception is that the **scratch** data structures need

to be distinct for each instance of the kernel. Typically an instance of the kernel is associated with each processor in the system. If each processor possesses sufficient private storage, each instance of scratch may be held in the memory associated with each processor. In the case of fully shared memory multiprocessors it is necessary to distinguish the private storage of different instances of the kernel by using an indexing scheme based on processor number to prevent the overwriting of the areas by other kernels.

As commercial SMP machines typically have uniform memory access speed, the organization of system tables has a limited effect on the operating efficiency of the kernel. In direct contrast to this, the performance of implementations on machines with NUMA (Non-Uniform Memory Architectures) organization is sensitive to the arrangement and location of system tables. On such systems it is necessary to place frequently accessed data in memory local to the processor. On architectures with a large communication path diameter, the organization of memory can become a critical factor in system performance.

To allow for efficient implementation of the kernel on multiprocessor systems with a NUMA organization, the Walnut Kernel has been designed to allow process information to be stored in memory local to the processor executing the process, and to minimize access to centralised tables. This approach leads to distributing system tables across the system. By associating system tables with individual processes, the Walnut Kernel provides a mechanism for decentralising the majority of the system tables and ensuring that kernel information required for a process can be easily identified and hence moved to memory local to the executing processor. A few centralized data structures are still required. Key among these structures are a list of objects currently in use in the system (AOT) and a list of processes to be run (mix). The kernel was designed to minimise the number of accesses required to these structures.

Multiprocessors with partitioned data in address spaces which are private to a processor or group of processors[9] such as the SP2 [AMM+95] can be supported by the Walnut Kernel. The SP2 is representative of a class of multiprocessors,

[9]Systems without a globally addressable memory space

which has recently become popular, known as a network of workstations. These systems communicate using message passing. The Walnut Kernel would implement distributed shared memory on this class of architecture by copying blocks of memory from address space to address space using the message mechanism provided by the hardware. Access to common tables would be performed by encapsulating the operations on the tables into messages and either broadcasting the message, where appropriate, or transmitting the message to the node managing the table. Sharing pages of user data could be performed by sharing read only copies of a page. The first process to attempt to write to a page would invalidate all other copies of the page and then allow the process to write to the page. A short time later the process would return the page to a read-only state and allow other processes to take new copies of the page. This mechanism allows processes to provide the semblance of shared memory on a partitioned data machine. However, the mechanism is expensive in both communications costs and page fault handling. This indicates that although the Walnut Kernel can operate in that environment it would incur significant overheads with programs using fine grained parallelism.

The practice of *timing out* data structures used by the kernel and the page mapping mechanism applied to a process have significant advantages over the alternative mechanism of recording processes that use a given capability and updating affected processes upon revocation of a capability. Although the *timing out* approach places a constant overhead on process execution this overhead is not directly dependent on the number or arrangement of the derivatives of an object's master capability nor is it dependent on the number of processes sharing a capability. The overhead of updating lists of processes using a given capability is especially burdensome on multiprocessing systems. This is because it requires the list to be locked during updates caused by processes loading and unloading capabilities. Contention over locks can cause major performance bottlenecks. The Walnut Kernel avoids this problem by ensuring that the structures describing a capability are only locked for a short time when a capability is being modified. This ensures that processes on separate processors sharing resources are not inhibited significantly by the loading and unloading of capabilities.

The expiring of page tables too rapidly introduces a significant overhead to a process, however, reducing the rate causes an increase in the length of time the rights conferred by a revoked capability persist. A balance between these competing interests determines the rate at which page tables are replaced. Noting that the cost of rebuilding page tables for a working set of pages is proportional to the number of active pages within the tables makes a rapid rate of expiration attractive as it reduces the amount of work required for the rebuilding of a table. A period of the order of a second is viable.

6.7.3 Messages

The message passing mechanism does not preserve the order of messages. In particular, when messages are passed between a single sender and a single receiver there is no guarantee that the messages will be received in the order sent. The property of preserving the order of messages is highly regarded by other system designers and is supported in systems such as the SP2 [SHFG95]. The presence of the property reduces the level of non-determinism present in parallel programs, enhancing the reproducibility of results.

Although the issues of preserving the order of messages were considered, the current design of the message passing mechanism makes no attempt to preserve chronological order. The decision was made on pragmatic grounds, as to properly support order of delivery it is necessary to have synchronized clocks for each processor to provide a precise representation of a universal time across the system. The requirements for precise timing would restrict the type of hardware that could be employed in contradiction of the design principles. Furthermore, encouraging programmers to believe that the order of messages is strictly preserved could promote unsafe programming practices in a multiprocessor system. Programmers might attempt to exploit the property using a collection of processes resulting in an increase in non-determinism of the system rather than a decrease in non-deterministic behavior.

When a Walnut Kernel process sends a message the process remains blocked until either the message is delivered or it is determined that the message cannot be

delivered. Another potential implementation is to block the process only until the message is queued for delivery to the other process. The alternative implementation is attractive as it allows a process to dispatch messages quickly and continue processing, however, it also has significant disadvantages. Among the problems is the design issue of the correct sizing of the queues to contain undelivered messages and the absence of a guarantee of delivery of a message. The current implementation does not suffer from the problems inherent to the alternative and although an individual process is delayed, other processes are able to use processor cycles while delivery is taking place. In addition, the Walnut Kernel implementation places greater control of the message mechanism in the hands of the programmer, as the programmer is able to determine what action to take if a message is deemed to be undeliverable.

In systems with a delivery buffer shared by a number of processes it is possible to create a deadlock by preventing the delivery of messages from a process to a process expecting that message. This potential is not present in the Walnut Kernel as each process has its own mail boxes. The ability to reserve mail boxes allows the programmer to guarantee the availability of a set of mail boxes for a given task. Control of the allocation of mail boxes gives the programmer greater control over message delivery allowing the programmer to manage resources to minimise or avoid the risk of deadlock.

6.7.4 Processes and Subprocesses

The original Capability Based Kernel did not support subprocesses and provided no mechanism for asynchronous event notification. The Walnut Kernel introduced subprocesses as a mechanism to allow asynchronous events to be handled by a process.

Subprocesses consist of a thread of execution through the process's address space. The Walnut Kernel supports up to 250 subprocesses within a process. Subprocesses are pre-emptively multitasked with the exception that they share access to a common parameter block and the subprocess claiming the parameter block excludes other subprocesses of the process from executing until the parameter block is released. Facilities for dealing with asynchronous events were added to processes by allowing

subprocesses to receive messages and reserve mail boxes for their use. A message constitutes an event asynchronous to the process. Subprocesses are necessarily co-operative by nature as they share a single address and share a critical common resource - the parameter block - and hence have no protection from each other. Processes may be adversarial in nature as they are immune from the effects of other processes unless address space is shared or another process possesses capabilities to the process or objects used by a process.

Only a single subprocess of a process may be active at one time. The design decision was driven by two considerations: ensuring compatible behavior between uniprocessor versions and multiprocessor versions, and avoiding splitting process data structures over the local memories of more than one processor. By eliminating multitasking at the subprocess level processes have compatible behavior on both single processor and multiprocessor systems. In NUMA machines, significant performance losses could result from a processor executing a subprocess with non-local process data structures when increased load on the communication mechanisms occurs.

The restriction on concurrent execution to processes encourages the use of subprocesses for their original function of supporting asynchronous events. The ability of a process to control the level of the exposure to the actions of other processes - by explicitly sharing address space - makes processes superior to subprocesses for many co-operative tasks.

6.7.5 Execute-Only Code

During the design process, it was observed that it would be desirable for the kernel to support the execute-only right found in the Monash Multiprocessor Project. As the primary target machine (Intel386) did not support execute right directly within page tables, the design option of introducing execute right by manipulating the page tables was explored. Several mechanisms for synthesising execute only code on machines with execute right implied by read right were proposed. The most time efficient mechanism was:

For each execute-only object loaded into the process's address space, a new set of top level page tables and private page tables is created. These page tables have read right set for the execute-only capability. All other entries in these page tables are set to invalid. The normal page tables used by the process have the invalid bit set for entries relating to execute-only objects.

Upon entry to an execute-only object a page fault occurs. The fault handler checks the Table of Loaded Capabilities and discovers that the fault location corresponds to an execute-only page. The fault handler substitutes the appropriate set of page tables for the normal set of page tables.

When an access is made outside the the execute-only area a page fault occurs. If the access is made to a non-execute-only capability the normal page tables are used, otherwise the appropriate set of page tables for the new execute-only capability are loaded.

The mechanism is quite expensive as it incurs the cost of a page fault for both entry to and exit from execute-only code and every memory access outside the execute-only capability. In addition the mechanism adds a significant number of overhead pages to a process using execute-only capabilities as it requires the presence of a number of sets of page tables.

The overheads required to provide execute-only code across a wide range of platforms were considered too high. Two motivations exist for the need for execute-only code. The first is the desire to hire out code to a user with the certainty that they cannot retain the use of the code after the hire period has ended. Unfortunately, the Walnut Kernel does not currently support a mechanism suitable for renting code, except by renting the services of a process which has access to that code. The second motivation is to hide a critical piece of information - typically a capability - within a code object. A new mechanism was introduced to provide an alternative solution to the problem of hiding critical data - the SRMULTILOAD right. SRMULTILOAD right allows capabilities to be derived which are only usable by a specific process, making dissemination of the capability unproductive.

A number of mechanisms have been considered for hiding the contents of an object. One mechanism requires a new class of objects known as *protected objects*. Protected objects are unreadable until a system-call - *enter* - is made which causes a jump to a fixed location in the object. The object is then made readable. A further system-call - *leave* - is used to make the protected object unreadable and return to the caller. Protected object mechanism is only a partial solution as it does not provide adequate protection in a process with several subprocesses and it does not protect the calling subroutine from the code contained in the called subroutine. The mechanism can be enhanced by making all other protected objects contained in a process unreadable when an enter call is made. This prevents called routines from spying on the caller. Further enhancement is required to deal with multiple subprocesses. Pages must be made readable or unreadable depending on which subprocess is executing. This is undesirable as it requires a process to maintain information about the accessibility of the object represented by each entry in the TLC for each subprocess. At present no acceptable mechanism has been found.

The prefered approach to making code widely available but unreadable is to encapsulate it in a Walnut Kernel process. Users wishing to employ the code can then request its execution by sending the process a message including capabilities for the data on which the code is to operate. Clearly, this approach is inefficient for code bodies with very short execution times, but these are unlikely to be candidates for hiring.

6.7.6 Hardware

The minimum requirements for supporting the Walnut Kernel on a system are:

- Support for paged virtual memory. This may be implemented using either page tables or translation look-aside buffers. If page tables are supported then a two or more level page table is required and page table entries must be of regular size and format having at least *valid* and *dirty* bits.

- A timer interrupt. A regular source of interrupts is required to allow pre-emptive multitasking to function.

- Integers of at least 32 bits in size must be available.
- Hardware support for atomic access to memory locations.
- Access to a time source of at least second accuracy.
- A privilege mechanism and a method for making system-calls.

These requirements are met by a wide range of currently available microprocessors from both the RISC and CISC design streams.

The Walnut Kernel places minimal requirements on the contents of page tables. It requires only the presence of a dirty bit and a valid bit (of course). The use of the Walnut Kernel's own data structures combined with the minimal requirements on a page table entry, enhance the kernel's portability. However, this comes at the cost of duplicating part of the data contained in the page tables. On systems with only translation look-aside buffers, such as the R4000 [MIP91], no duplication occurs.

6.8 System Initialization

The system initialization process is unique within the Walnut Kernel as although it is persistent (as are the other processes) it is restarted each time the kernel is started. The capability mechanism enhanced with the restrict operation allows user processes to manipulate the system memory with high security. The initialization process is run as an ordinary user process. The only special feature of the initialization process is that no other process can execute before the initialization process allows the process scheduler to start. It uses system-calls to derive relatively safe capabilities for user level device drivers from a capability for the kernel address space. The initialization mechanism allows the existing derivation code to be employed minimizing the amount of specialised code for use only during initialization found within the kernel.

Currently the initialization process performs the following tasks:

1. read the capability for the system memory from the wall (a page mapped into the address space of all processes)

2. derive capabilities required by device drivers

3. restrict the capability for the system memory

4. remove the capability for the system memory from the wall

5. start the process scheduler

6. send messages to device drivers containing the required capabilities

The scheduler is enabled to schedule processes other than the initialization process by writing into an address in the kernel area of the system memory. By ensuring that other processes are not scheduled before the initialization process has restricted and removed the publicly readable version of the system capability, leakage of information critical to the function of the system to malicious processes is prevented.

The current system uses a constant for the value of the capability for the physical object. However, by having the initialization process read the value of the capability from the wall, a randomly selected set of passwords could be employed without changes to the initialization process.

6.9 Money

This section addresses the issues of rent collection and payment for kernel services. The mechanisms discussed have yet to be implemented in the experimental version of the kernel.

The rent collector periodically scans every process on the mounted volumes and deducts rental proportional to the time since rent was last collected from the object. The rent collection mechanism allows for variations in the rate of operation of the rent collector and for consistent charging in proportion to resource use for off-line volumes. Rent is collected from an object's store of money. Bankrupt objects are deleted.

The system-call interface incorporates a charge which is levied against the cash of the process making the call. Although it would be desirable to levy the process for the amount of work required to complete the system-call, this would introduce uncertainty for the caller and complicate the collection process.

6.10 Process Scheduler

The Walnut Kernel's existing round-robin process scheduler is suitable for testing purposes in an experimental implementation, but is inadequate for a production system. In a mature system, a more sophisticated scheduler would be employed. This scheduler would use a round robin policy for a set of processes with current wakeup times (the process is scheduled to wake up at or before the current time). A set of lists of processes with access times in the future would be maintained. The lists would contain processes scheduled for wakeup within a time period. Periodically each list would be examined and elements would be migrated to appropriate lists or the current queue. Figure 6.17 illustrates the proposed scheme.

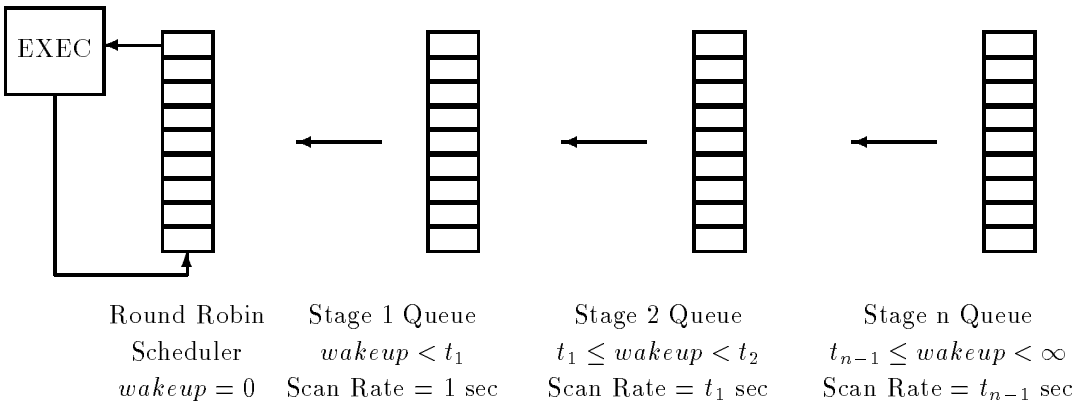


Figure 6.17: Proposed Scheduling Scheme

It is intended that this scheme will be implemented in two parts. The first will consist of the existing round robin scheduler within the kernel but enhanced so that the interface to the queue will be accessible to user processes holding an appropriate capability. The second part will be one or more user level processes which will maintain the lists of processes, and move processes into the *mix*.

The proposed mechanism allows process scheduling to be conducted at the user level. User level scheduling offers the possibility of exploring alternative near term and long term scheduling schemes, and simplifies the kernel. Short term scheduling is retained in the kernel for the purposes of efficiency, to eliminate the need to communicate with a user process on each process swap.

The current implementation of the process scheduler is clearly inadequate for a

production system. The highly restrictive limits it places on the number of processes is inappropriate for a persistent based system. However, the implementation allowed the system to be assembled and tested quickly.

The proposed scheduler addresses the deficiencies of the current implementation.

Chapter 7

Implementation

At present there are two implementations of the Walnut Kernel. The first of these runs under both UNIX and MS-DOS[1]. This version simulates the hardware supporting the kernel. The second version runs on IBM-PCs using Intel386 and i486 processors based on an ISA bus architecture. This chapter describes the two implementations and the mechanisms used to load user programs onto a native system.

7.1 The Standalone Implementation

The Standalone version of the Walnut Kernel emulates aspects of the underlying hardware and executes above a host operating system. Its name has been the subject of debate and is sometimes considered a misnomer. It was named the Standalone version early in the development to imply that it did not require the support code used in a native version. The name is somewhat misleading as the Standalone version is dependent on the host operating system for persistent storage, memory and IO. This version does not support the execution of compiled programs. Instead, a special type of process known as *drive processes* are executed.

Drive processes are interpreters for a simple language that allows the contents of the parameter block to be specified and system calls to be invoked. The language has constructs that allow for iteration and alternation providing a language sufficiently powerful for testing all the higher level functions of the kernel.

[1]MS-DOS is a trademark of Microsoft Corporation

This version of the kernel shares all the architecture independent code of the kernel with the exception of the process scheduler which has been modified to support the operation of drive processes and the system call mechanism. The scheduler, in the standalone version, is a loop which selects the next process to run, sets up the parameter block, and invokes the drive process interpreter. When the drive process returns, the 'program counter' - position in the drive process program - is stored and the loop is repeated. The drive process does not access the system call mechanism, instead it calls the routines that the system call mechanism would use directly.

The system memory controlled by the Walnut Kernel is simulated by allocating a large block of memory when the kernel is started. This is then divided up by the support routines before the scheduler is invoked. Accesses to memory by the drive process are simulated by having the drive process call a subroutine which accesses the page tables constructed by the kernel in the simulated memory. These routines emulate the activities of access which result in page faults as well as ordinary accesses.

A single volume is simulated by the supporting software. It is represented as a memory array. This volume can be loaded from or stored to disk. The user is prompted for a file, when the kernel starts, from which to read the disk image and has the option of specifying a file name for storing the image when the system is shutdown. The disk images are accurate representations of the data stored in disk blocks on a native system. These images are used as one of the mechanisms for loading programs into a native system (see section 7.3).

The Standalone system was invaluable in the development of the Walnut Kernel as it permitted the development of the kernel in an environment where debugging facilities where available. Activities of the kernel could be logged to files and repeatable test scripts could be run. Early in the development of the i486 version several significant difficulties occurred: triple faults could occur which resulted in a reboot; timing problems resulted in subtle, difficult to repeat errors; and there was no permanent store available for logging. The Standalone version provided both a mechanism to build disk images used to initialize the native system, and a testing ground that allowed the high level routines to be exercised in a friendly environment

before attempting to use them in a native system.

The Standalone version of the kernel has been used in the following environments:

- 80386 based PC running DOS
- 80486 based PC running OS/2
- 80486 based PC running FreeBSD
- R3000 based workstation running Ultrix
- R4000 based workstation running Irix

7.2 i486 Implementation

The current implementation of the Walnut Kernel runs on Intel386 and i486 based personal computers. The kernel makes direct use of the system hardware, requiring only the boot sector loader and initialization code contained in the PC BIOS.

Use of an existing operating system as a host allows the implementor to use the work of the designers of the host operating system. However, the price is a loss of flexibility, and the additional overhead of accessing the host operating system. For these reasons the kernel was made to rely only on the system hardware for support. The option of using MS-DOS as a host was rejected. In addition, only the loading and initialization services provided by the BIOS were used as the standard PC BIOS routines are written to operate in the processor's real mode (16-bit mode) only.

The i486 supports a number of architectural features which may be exploited at the operating system level[Int90, CG87]. These features include:

Segments - Variable length segments are provided as the primary mechanism for memory protection. This mechanism enables checking of segment type (readable, writable, and executable), segment limits, and segment privilege levels.

Paging - Two level page tables with user/supervisor and read/write bits are supported. Translation lookaside buffers are loaded automatically by the processor using the contents of the page tables.

Multiple Privilege Levels - Two distinct privilege mechanisms are present: a 4 level segment based mechanism and a 2 level page based mechanism.

Tasks - A task data structure is required by the processor. This data structure contains the values of registers for a task at each of the privilege levels supported by the processor.

I/O space - Access to the I/O memory space of the processor can be controlled on a segment based privilege level basis or on a per task basis using a map which allows access to I/O addresses selectively.

The current system uses Intel's 'Flat Model' of memory management. Under this model, segments are present in a minimal capacity, and cover the complete address space of the processor. The paging mechanism is used to provide protection and a large virtual address space. This decision conforms to the requirements of section 6.1.1 where only features available on a wide class of processors are exploited by the Walnut Kernel. A two level privilege scheme is the most general mechanism for providing the necessary protection for the kernel. This mechanism was adopted in the interests of portability.

The task mechanism and its associated I/O address space access management allowed the Walnut Kernel to have user level processes with the ability to operate as low level device drivers. A scheme was adopted using a single task with multiple user processes and a single kernel shared by the user processes. The current scheme has the advantages of greater portability offering similar levels of support to those found on RISC processors. It also avoids the need for the kernel to manage multiple task state segments.

I/O address space access was limited to the kernel to force low level drivers into the kernel. User level low level drivers have the apparent 'advantages' of being modifiable during the operation of the system and having the same facilities available as user level programs. User level drivers can complicate the support of user processes as they place new and significant demands on the kernel. The ability to allow selective access to hardware facilities is required for user level drivers. In systems with memory mapped I/O this is clearly compatible with the concept of capabil-

ities. Supporting systems with a separate I/O address space requires additional complexity to be added to the capability model. This is because there is no longer a single uniform resource consisting of memory to be managed, but two distinct resources, memory and I/O address space to be managed. The user level driver approach is further complicated by the need to ensure that drivers have adequate access to physical memory and execution time. Lack of either of these resources could result in a deterioration of system performance. By placing device interrupt handlers into the kernel, it is possible to make the kernel responsible for the correct operation of the system. This allows the kernel to ensure that accesses to hardware do not place the system into an unstable state. Moving low level drivers into the kernel allows scheduling and resource issues to be handled easily. As a result of these considerations, the Walnut Kernel implements device interrupt handlers within the kernel.

The kernel does not use many of the special features of the Intel386 and i486 to ensure a simple migration path from the initial implementation to generic hardware.

i486 implementation currently has low level device drivers for RS232 serial ports, Centronics parallel ports, keyboard, and color and monochrome text based displays. The circular buffers used to access these devices are held in the first 16 pages of memory. Access to volumes (block oriented devices) is provided by a low level device driver which supports ST506 and IDE based hard disks. A floppy disk driver is provided for reading programs from DOS formatted disks.

7.3 Loading Programs into the Walnut Kernel

There are three mechanisms currently available for loading executable programs into the Walnut Kernel:

- Using a serial port under the control of a drive process.
- Adding a program to the collection of objects written to the boot drive when the system is initialised.
- Using Shell (see section 9.6) to read the program from a DOS formatted floppy disk.

The Walnut Kernel expects executable programs to consist of two parts. The first part is the 'text' component (the instructions to be executed), and the second part is the 'data' component (static data, global variables, etc). Constants can be placed in either part. Two objects are used so that the program data can be copied and hence used for each each process without interference, and the program code can be shared. Programs are currently compiled under *gcc* and linked with *ld*. A program reads the resulting **a.out** file and produces a text and a data file. These are then transfered to the Walnut Kernel using one of the techniques described below.

The i486 version of the Walnut Kernel currently supports *drive processes*. These are used as a debugging aid and for testing purposes. The two serial ports provided in the current version are used to provide a console - operating on a glass tty - for the drive processes, and a serial IO stream used for transferring data. A primitive communication protocol is used on the serial connection to drive a program running under DOS. The program reads disk files and transfers them through the serial connection. Although this mechanism is still present in the kernel, it is no longer used. The simple protocol was prone to failures caused by the loss of synchronization between the sender and receiver; at the time the inconvenience caused by this was not sufficient to warrant the additional effort required to improve the protocol. Other mechanisms have been implemented which are faster and more reliable.

When a bootable system is being created, the initialization process and the data it operates on must be transfered to the system. These objects must be present before the system is started to ensure that they are placed in standard locations and hence can be found by both the kernel and the initialization process. To perform this task, the standalone version of the kernel is employed, to load the objects in a prescribed order and create a disk image file. The volume and serial numbers are known for each object created by the standalone version because: the size of each object is known, the order of loading is specified, the algorithm for allocating disk blocks is known and no objects are removed. Furthermore, the series of instructions used by the standalone version to load the object from disk includes instructions to store the master capability of the objects loaded in an object on the disk. This object is read by the initialization process on startup and then deleted. Other user

programs can be added by having the standalone version load their code and data objects. The master capability for the objects created can be displayed using a drive process. The final disk image file produced by the standalone version is read by one of the install programs, and the blocks are copied to the install disk.

A facility to read files from DOS formated floppy disks and make objects under the i486 version of the kernel was introduced into the shell. This mechanism uses the floppy disk manager to access the floppy disk at the sector level. Shell has functions which allow it to read files from disk and read the disks directory.

Chapter 8

Performance

This chapter details a number of performance measurements carried out on the Walnut Kernel. The method of the experiments and the conditions under which they were conducted are discussed. In addition, the performance of a conventional UNIX operating system, operating on similar hardware, is provided to permit some comparison to be made between the Walnut Kernel and a conventional operating system. The tests conducted under UNIX approximated the functions of the Walnut Kernel. However, significant differences in design between UNIX and the Walnut Kernel, prevent a close comparison of the performance of the two systems.

8.1 Test Environment

8.1.1 Software

The tests were conducted on a version of the Walnut Kernel which did not contain the kernel debugger but retained diagnostic code. This version of the kernel had not been optimised for performance.

The tests on UNIX were run on a FreeBSD[1] V1.1R operating system [Wal94]. This version of UNIX is a derivative of the University of California Berkeley's Networking Release 2 of 4.3BSD.

Note that the Walnut Kernel was compiled without optimisation, whereas the

[1]The FreeBSD CDROM is a trademark of Walnut Creek CDROM

UNIX kernel was highly optimised.

8.1.2 Hardware

The test machines were as close to identical as possible in that they used motherboards, peripherals and interface cards sourced from the same manufacturer. The critical parameters of the test machines were specified as follows:

Intel 486DX 33MHz

8 Mb RAM

Caviar 2120 HDD

Both the 80486DX's internal cache and the external 128k cache were disabled for the tests. The caches were disabled to make the initial conditions of all experiments as similar as possible. In addition, because of the wide variety of external caches available for the i486 and the difficulty in determining the type of cache in use from system specifications, disabling the external cache would tend to improve the repeatability of results when the experiments are performed on similarly specified hardware.

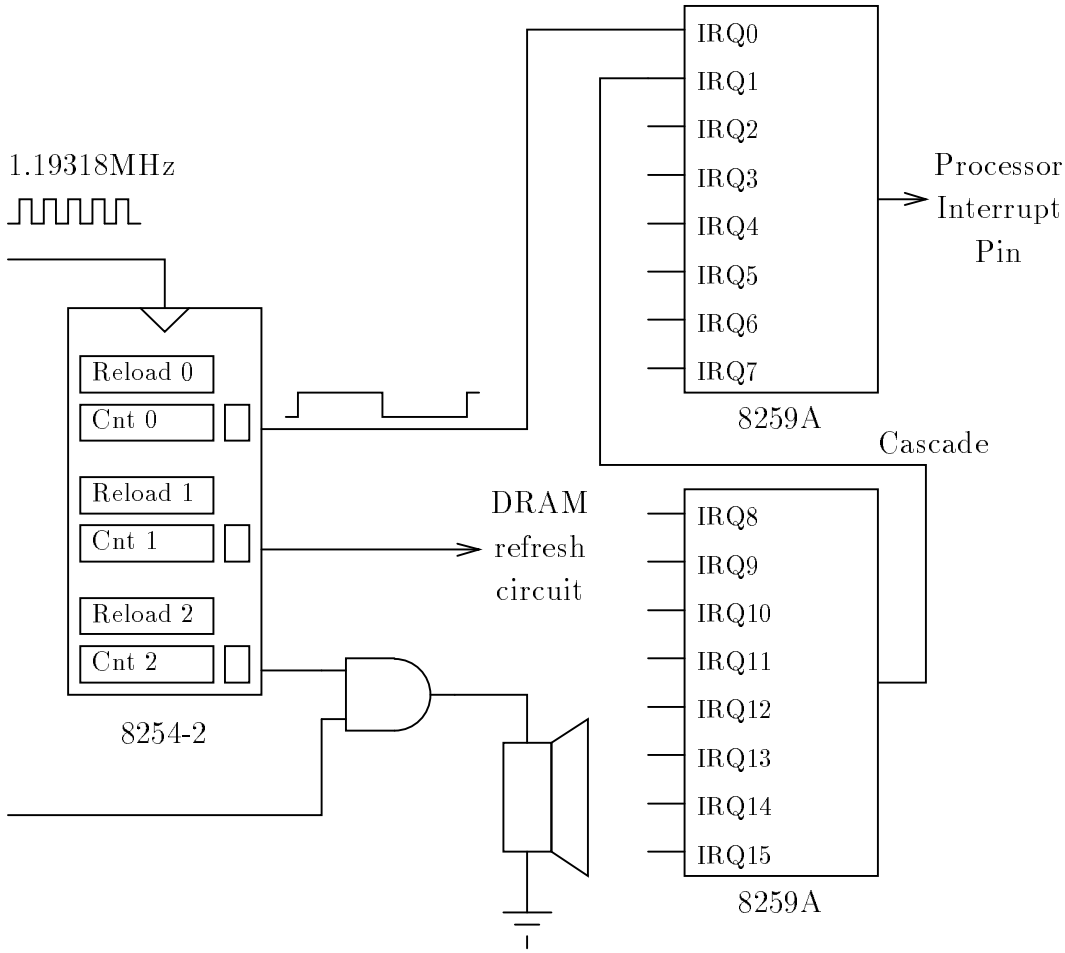
There is no apparent reason to suggest that the memory reference patterns of the Walnut Kernel would be less favorable for caching than UNIX.

8.2 Timing

The test programs on the Walnut Kernel had access to a 32 bit counter that is driven by a 1.19318 MHz frequency source. This counter was derived from channel 0 of the system's Programmable Interval Timer [SC90, Mac84, Tri92, Nor85]. High resolution timing - The resolution of the timer is ~840nS - is achieved by polling the Programmable Interval Timer's counters [Rod92].

The 8254-2 Programmable Interval Timer supports 3 channels. Although several modes of operation are supported by each of the 8254-2's channels, only the square wave generator mode is relevant to the implementation of high resolution timer. When configured as a square wave generator, a channel provides a 16-bit counter which is driven by the 1.19318 MHz frequency source. The 16-bit counter is loaded

with the *reload value*, and the counter proceeds to count downwards by two. Each time the counter reaches zero, the output bit of the channel is inverted and the counter is reloaded with the reload value. The output bit of a channel in this mode, with an even reload value, generates a square wave with a 50% duty cycle.



8254-2 - Programmable Interval Timer
8259A - Programmable Interrupt Controller

Figure 8.1: Timer and Interrupt Hardware in the IBM-PC/AT

In the IBM-PC/AT architecture (see figure 8.1): channel 0 is used to provide timer interrupts; channel 1 is used to provide memory refresh cycles, and channel 2 is used as a tone generator for driving the system speaker. Channel 0's output bit drives *interrupt request* line 0 (IRQ0) which is connected to the interrupt controller.

The interrupt controller generates a timer interrupt[2] on each positive transition of IRQ0.

As the 16-bit counter provided by the 8254-2 is decremented by twos, the least significant bit of the timer's counter provides no useful information. A 16-bit counter is synthesised by extracting the 15 useful bits from the timer's counter and using the timer's output bit to provide the most significant bit of the synthesised counter value.

The process scheduler is entered whenever a timer interrupt occurs or a process surrenders its time-slice. The process scheduler determines the number of counts that have elapsed between the current access to the timer and the last access, and adds this to a 32 bit counter that is visible to programs. This guarantees that whenever a process's code is entered the current value of the 32 bit counter is accessible.

The microsecond timing provided by FreeBSD is derived from the channel 0 timer of the Programmable Interval Timer using a method similar to that described above. The *gettimeofday()* library function is used to access the time.

8.3 Walnut Kernel

Three programs were written to test the performance of the Walnut Kernel. The programs contained several tests. Each test consisted of 100 repetitions of a set of system calls. The results of the tests were written into an object created by the test program.

A 25 second delay was built into the test program to allow the *drive*-type[3] process to be used to *freeze* any other processes present on the system and to allow time for the system to settle after performing this task.

[2] Timer interrupts are known as *ticks*

[3] **Drive** processes are processes which execute a simple command interpreter built into the kernel. Drive processes are intended only to be used in the development phase of the Walnut Kernel and will be removed from other versions of the kernel

8.3.1 Program 1

This program tested the performance of communication related system calls. Specifically it tested the speed of the following system calls:

- external send
- send
- receive
- load capability
- unload capability

Test 1 and Test 2

These tests measured both the time taken to execute an *external send* system call and the total time taken to complete the transfer from the beginning of the first attempt until the end of the successful attempt. In both tests no money was transferred between the sender and the receiver. **Test 1** had a message of zero length and **Test 2** had a message of 64 bytes length.

Test 3 and Test 4

These tests measured both the time taken to execute a *send* system call and the total time taken to complete the transfer from the beginning of the first attempt until the end of the successful attempt. The target process was loaded into a large window of the test process and identified by *offset*. In both tests no money was transferred between the sender and the receiver. **Test 3** had a message of zero length and **Test 4** had a message of 64 bytes length.

Test 5 and Test 6

These tests measured both the time taken to execute an *external send* system call and the total time taken to complete the transfer from the beginning of the first attempt until the end of the successful attempt. A *wait* of one second was introduced

between each repetition of the test to ensure that the target process had an empty mailbox available to receive the message. In both tests no money was transferred between the sender and the receiver. **Test 5** had a message of zero length and **Test 6** had a message of 64 bytes length.

Test 7 and Test 8

These tests measured both the time taken to execute a *send* system call and the total time taken to complete the transfer from the beginning of the first attempt until the end of the successful attempt. The target process was loaded into a large window of the test process and identified by *offset*. A *wait* of one second was introduced between each repetition of the test to ensure that the target process had an empty mailbox available to receive the message. In both tests no money was transferred between the sender and the receiver. **Test 7** had a message of zero length and **Test 8** had a message of 64 bytes length.

Test 9 and Test 10

This test measured both the total time taken to transfer a message and the time taken to execute a *receive* system call. The transfer time is calculated from the beginning of the first attempt to perform a send to the time at which the contents of the message is made accessible by a successful *receive* call. A message of 4 bytes length and with no money was sent. A *wait* of one second was introduced between each repetition of the test to ensure that the target process had an empty mailbox available to receive the message.

Test 9 used an *external send* system call to perform the transfer. **Test 10** used a *send* system call to a process loaded into a large window.

Test 11 and Test 12

These tests measured both the time taken to *load* and the time taken to *unload* an object from a window of the test process's address space. **Test 11** used a large window and **Test 12** used a small window.

8.3.2 Results

The following table summarises the raw results for each experiment. The measurements are in microseconds.

Messages

	High	Low	Average	Median
Exp 1 - 0 byte transfer				
K_EXTSEND	782.78	458.44	482.55	477.72
Total time	5558.26	471.85	755.38	489.45
Exp 2 - 64 byte transfer				
K_EXTSEND	812.95	460.11	496.79	496.99
Total time	3286.18	492.80	752.45	500.34
Exp 3 - 0 byte transfer				
K_SEND	880.84	585.83	610.72	605.94
Total time	3095.93	600.92	913.76	613.49
Exp 4 - 64 byte transfer				
K_SEND	882.52	584.99	622.11	624.38
Total time	2567.93	620.19	919.03	628.57
Exp 5 - 0 byte transfer				
K_EXTSEND	511.24	497.83	504.71	505.37
Total time	511.24	497.83	504.71	505.37
Exp 6 - 64 byte transfer				
K_EXTSEND	533.03	517.11	525.89	526.32
Total time	533.03	517.11	525.89	526.32
Exp 7 - 0 byte transfer				
K_SEND	993.14	616.84	742.84	626.06
Total time	993.14	616.84	742.84	626.06
Exp 8 - 64 byte transfer				
K_SEND	1013.26	636.12	762.22	645.33
Total time	1013.26	636.12	762.22	645.33
Exp 9 - 4 byte transfer				
Ext. Trans	813.79	794.52	805.00	805.41
K_RECV	301.71	298.36	299.84	300.04
Exp 10 - 4 byte transfer				
Send Trans	1294.86	888.38	1025.80	899.28
K_RECV	306.74	273.22	284.51	274.90

Address Space Management

	High	Low	Average	Median
Exp 11				
Load Large	3118.56	740.88	779.63	755.96
Unload Large	885.03	621.03	631.12	626.06
Exp 12				
Load Small	995.66	690.59	711.31	706.52
Unload Small	652.88	633.60	641.66	638.63

8.3.3 Program 2

This program tested the speed of system calls which manipulate objects. The following system calls were tested:

- Make Object
- Bank
- Destroy Capability

Test 1 to 4

This test measured the time required to execute a *make object* system call, to perform a deposit and a withdrawal on that object using the *bank* system call, and to destroy the object using a *delete capability* system call with the master capability of the object. There was a 1 second delay between repetitions of this test.

Test 1 generated objects of a single page in size. **Test 2** made a 2 page objects, **Test 3** made 4 page objects and **Test 4** was applied to 8 page objects.

8.3.4 Results

The following table presents the raw results for each experiment. The measurements are in microseconds.

Object Management

	High	Low	Average	Median
Exp 1 - 1 page object				
K_MAKEOBJ	800.38	613.49	710.73	708.19
Deposit	515.43	489.45	502.35	502.02
Withdraw	374.63	348.65	359.49	356.19
K_DEL	341.11	319.31	330.78	329.37
Exp 2 - 2 page object				
K_MAKEOBJ	903.47	673.83	795.79	810.44
Deposit	515.43	489.45	503.70	503.70
Withdraw	373.79	349.49	360.45	362.06
K_DEL	341.94	319.31	330.44	329.37
Exp 3 - 4 page object				
K_MAKEOBJ	1002.36	611.81	870.35	911.85
Deposit	520.46	493.64	504.54	502.86
Withdraw	373.79	349.49	360.41	358.71
K_DEL	342.78	319.31	330.24	328.53
Exp 4 - 16 page object				
K_MAKEOBJ	2053.34	709.87	960.45	999.01
Deposit	531.35	494.48	508.59	509.56
Withdraw	372.95	347.81	361.32	364.57
K_DEL	348.65	320.15	330.47	331.89

8.3.5 Program 3

This program tested the performance of process related system calls. Specifically it tested the speed of the following system calls:

- make process
- bank
- unload capability
- delete capability

Test 1

This test measured the times required to execute a *make process* system call, to perform a withdrawal on that process's object using the *bank* system call, to unload the new process from the test process's address space, and to destroy the process

using a *delete capability* system call with the master capability of the process. There was a 2 second delay between repetitions of this test.

8.3.6 Results

The raw results for each experiment are in the following table. The measurements are in microseconds.

Process Management

	High	Low	Average	Median
Exp 1				
K_MAKEPROC	104149.42	74981.98	76018.40	75020.53
Withdraw	377.98	354.51	367.82	367.92
K_UNLOADCAP	641.98	623.54	632.44	632.76
K_DEL	346.13	323.51	336.53	336.91

8.4 UNIX

Two programs were written to perform measurements under UNIX. The first program contained several separate tests. The second program contained a single test routine. Each test consisted of 100 repetitions of a set of library calls. The results of the tests were written into a number files created by the test program.

The programs were run on a FreeBSD system which had just been rebooted and had only a single interactive session operating on it. This session was used to run the two test programs.

8.4.1 Program 1

This program evaluated the times taken to perform interprocess communication tasks, file system tasks and memory allocation tasks.

Test 1 and Test 2

These tests used two processes connected by a pipe to test the time required to send and receive messages. After creating a pipe, the test process created a child process through the use of a fork system call. The child sent messages to the parent

process through the pipe. The times spent in the *write* and *read* library functions were recorded. Both the *write* and *read* calls were operated in non-blocking mode.

Test 1 had a message of zero length and **Test 2** had a message of 64 bytes length.

Test 3 and Test 4

These tests used two processes connected by a pipe to test the times required to send and receive messages. After creating a pipe, the test process, created a child process through the use of a fork system call. The child sent messages to the parent process through the pipe. The times spent in the *write* and *read* library functions were recorded. A one second delay between attempts to transmit a message was introduced. Both the *write* and *read* calls were operated in non-blocking mode.

Test 3 had a message of zero length and **Test 4** had a message of 64 bytes length.

Test 5

This tests used two processes connected by a pipe to test the time required to send and receive messages. After creating a pipe, the test process, created a child process through the use of a fork system call. The child sent messages to the parent process through the pipe. The time taken from entering the *write* library function to exiting the *read* library function was recorded. A one second delay between attempts to transmit a message was applied. Both the *write* and *read* calls were operated in blocking mode.

Test 6 and Test 7

Test 6 measured the time required to create a zero length file using *fopen* and the time required to destroy the file using *unlink*.

Test 7 measured the time required to open an existing file using *fopen* and the time required to close the file using *fclose*.

Tests 8 to 15

Tests 8 to **11** measured the time required by *malloc* to allocate space and *free* to release the allocated space. The tests allocated 1, 2, 4, and 16 pages of space respectively.

Tests 12 to **15** measured the time required by *calloc* to allocate space and *free* to release the allocated space. The tests allocated 1, 2, 4, and 16 pages of space respectively.

8.4.2 Results

The following tables show the raw results for each experiment. The measurements are in microseconds.

Inter-Process Communication

	High	Low	Average	Median
Exp 1 - 0 bytes				
read	93156	296	1235.21	297
write	1385	569	592.35	572
Exp 2 - 64 bytes				
read	1054	296	495.17	645
write	15902	569	877.45	677
Exp 3 - 0 bytes				
read	816	296	313.80	297
write	16528	580	740.60	581
Exp 4 - 64 bytes				
read	4740	519	1403.62	521
write	15952	566	722.30	568
Exp 5 - 8 bytes				
transfer	16958	999	1162.52	1003
write	24047	1750	2074.98	1815

File Management

	High	Low	Average	Median
Exp 6 - Create File				
fopen	15146	14497	14693.86	14654
unlink	26102	23076	23719.08	23732
Exp 7 - Open Existing File				
fopen	1410	874	911.27	877
fclose	1208	463	482.72	465

Memory Management

	High	Low	Average	Median
Exp 8 - 1 page				
malloc	2985	1178	1244.34	1187
free	616	140	149.74	142
Exp 9 - 2 pages				
malloc	1923	1196	1255.79	1205
free	378	140	146.62	142
Exp 10 - 4 pages				
malloc	2864	2043	2307.17	2296
free	691	140	151.66	142
Exp 11 - 16 pages				
malloc	108099	2457	3806.77	2731
free	396	140	146.83	142
Exp 12 - 1 page				
calloc	64795	1020	1710.56	1028
free	587	140	145.47	141
Exp 13 - 2 page				
calloc	180489	1870	5304.91	1881
free	867	139	150.68	141
Exp 14 - 4 page				
calloc	4046519	3575	63252.07	3610
free	866	140	148.58	141
Exp 15 - 16 page				
calloc	3134522	16196	336518.51	102325
free	65845	141	8146.38	143

8.4.3 Program 2

This program evaluated the times taken to create a new process.

Test 1

This test used a *fork* library call to generate a child process which performed an *exit*. The time required by the parent process to perform the *fork* was measured.

8.4.4 Results

The following tables list a summary of the raw results for each experiment. The measurements are in microseconds.

Process Management

	High	Low	Average	Median
Exp 1				
fork	47191	11294	14263.85	11331

8.5 Observations

Making direct comparisons between the Walnut Kernel and FreeBSD is difficult as the two architectures support different paradigms and operations. The Walnut Kernel's memory object paradigm is not reproduced in UNIX. Operations on objects have some of the characteristics of operations on files - persistence - and other characteristics best modeled by UNIX's memory management libraries. A major difference, in operation, between the two systems was the method of acquiring the current time. Under FreeBSD a system call was required to get the current time whereas under the Walnut Kernel the current time was available as a by-product of any system call. The use of intrusive measurement on UNIX and non-intrusive measurement on the Walnut Kernel made the duplication of some of the behaviors of the test programs impractical. As a result, only broad comparisons can be made between the systems.

8.5.1 Walnut Kernel Behavior

Every 10 seconds a process is required to rebuild its private and top level page tables and the process also invalidates all its windows. When the process next accesses a memory location the capabilities for that window is revalidated. Introducing a delay between tests reduces the number of operations per second and increases the effect of these overheads per operation performed.

8.5.2 Messages & IPC

An unexpected feature of the measurements of the Walnut Kernel's message delivery system was that K_EXTSEND performed better than K_SEND. This raises the possibility of improving the message transfer performance of the system by optimising K_SEND. The transfer of data to a process which is already loaded into the sender's

address space should require fewer checks on the accessibility and validity of the target process. In addition, the transfer of data should be simplified as the target page is loaded into the processes address space.

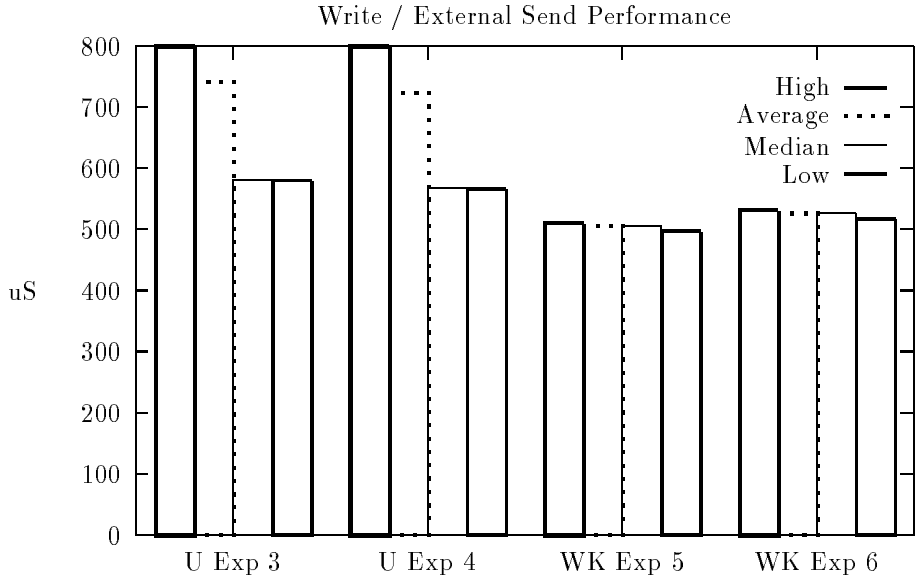


Figure 8.2: Comparison of External Send to Write Operations

Figure 8.2 compares the K_SEND and *write* times for the two systems. The message transfer performance of the two systems is similar. The graph indicates that the performance of the *external send* operation is typically faster than that of the *write* operation, and that the worst cases encountered for the Walnut Kernel were significantly better than those encountered by FreeBSD.

Figure 8.3 compares the time taken to complete a transfer of data between two processes. The Walnut Kernel performs significantly better than FreeBSD in this test. The worst case performance of IPC using pipes under UNIX is significantly worse than for the equivalent operation under the Walnut Kernel.

8.5.3 Address Space Management & File Management

The typical performance of *fopen* and *load capability* operations to be similar. as shown in figure 8.4. However, the worst case performance of the Walnut Kernel *load capability large* operation is approximately twice the cost of the worst case of *fopen*.

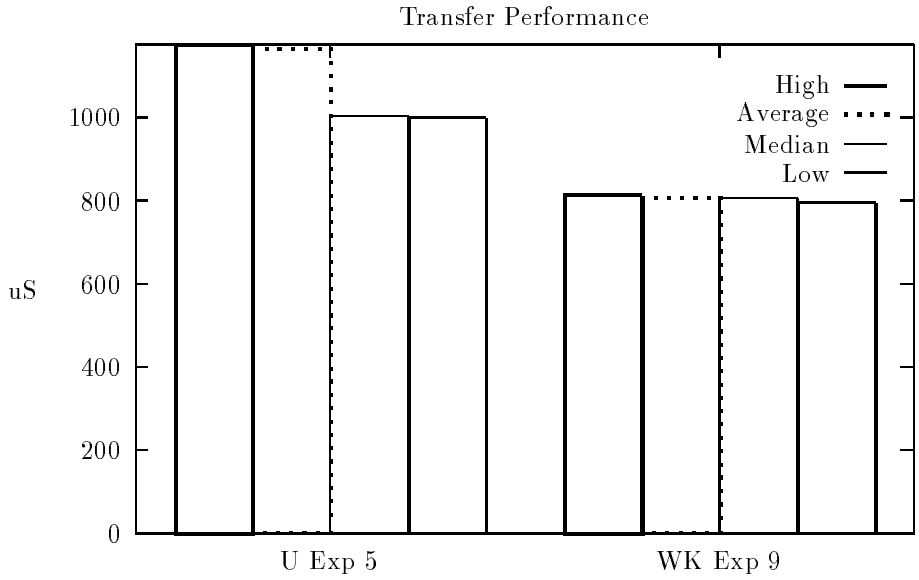


Figure 8.3: Comparison of Transfer Time Between Two Processes

Although the typical speed of performing an *unload* operation is about 30% larger than that of *fclose*, the worst case performance of the Walnut Kernel is better than UNIX.

8.5.4 Object Management & File Management

The creation of objects under the Walnut Kernel is approximately equivalent to the creation of files under UNIX as objects and files represent the units of persistent storage found in the two systems. Figure 8.5 compares the performance of the two systems in this area.

Although the time required by the Walnut Kernel to create an object is dependent on the size of the object and the initial number of capabilities required, object creation time seems to be significantly faster than file creation time under FreeBSD.

The destruction of objects is approximately equivalent to the deletion of files under UNIX, as both destroy items in the persistent store. In figure 8.6 the Walnut Kernel is shown to perform better in terms of returning more quickly than FreeBSD. However, it should be noted that although this prevents new Walnut Kernel processes loading the object immediately, processes which currently have the object

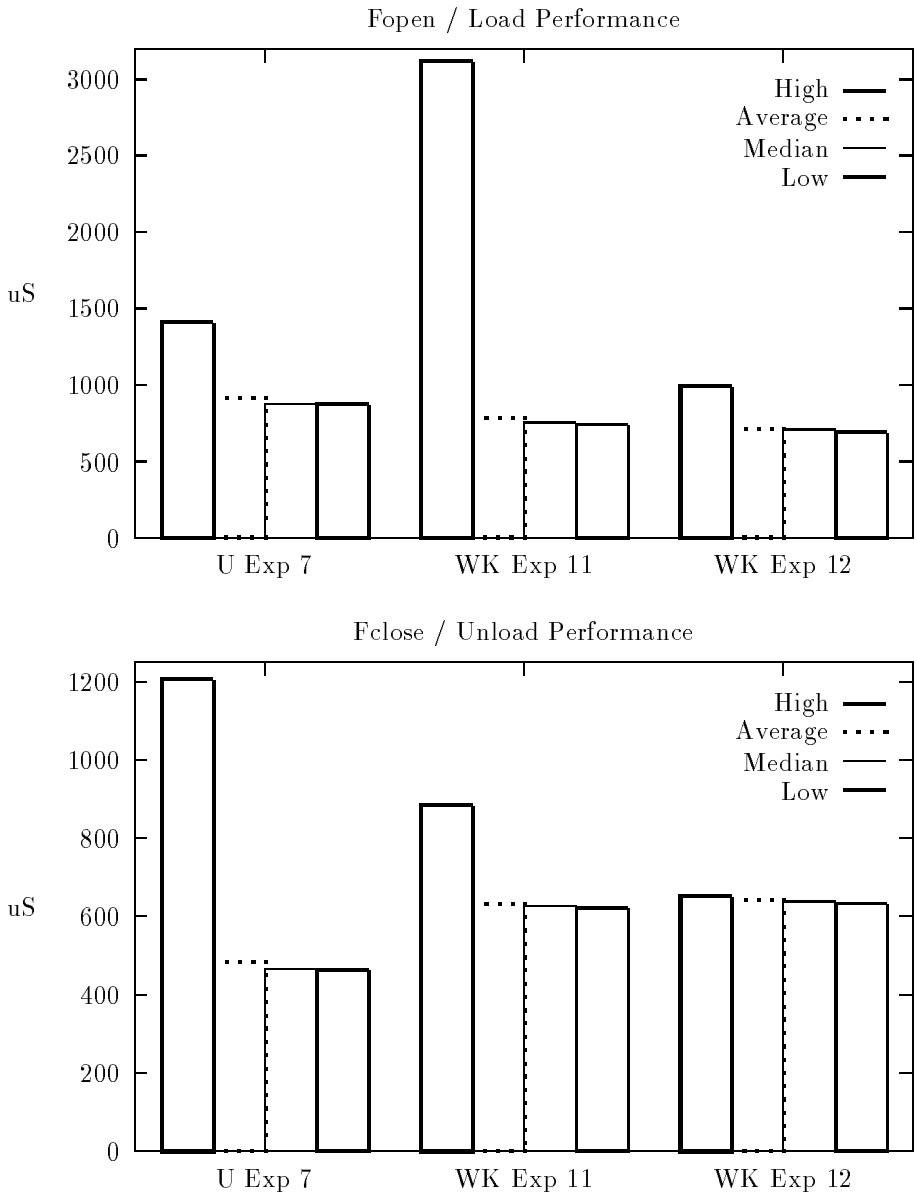


Figure 8.4: Comparison of File Operation Times to Object Operation Times

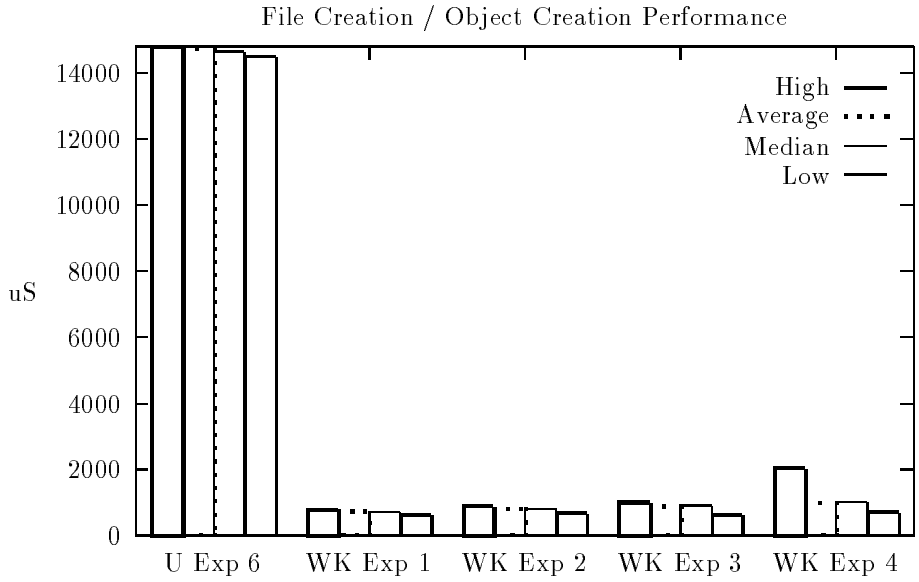


Figure 8.5: Comparison of File Creation Time to Object Creation Time

loaded will retain access for 3 seconds.

8.5.5 Process Management

The *fork* operation to the *make process* operation are compared for purposes of completeness. The comparison yields little useful information as the result of a *fork* is to produce a copy of an existing process, whereas the *make process* operation produces a new process. The closest approximation of the Walnut Kernel's *make process* offered by UNIX is a *fork* followed by an *exec*. Unfortunately as the exec system call does not return, it is not possible to measure the speed of that system call using user level programs. Figure 8.7 compares the two operations.

8.6 Conclusion

Where operations were comparable, the Walnut Kernel performed as well or better than the competing FreeBSD system. The Walnut Kernel's poorest performance, when compared to UNIX, was in the generation of new processes. However, this comparison is of limited significance: as the Walnut Kernel generates a new process

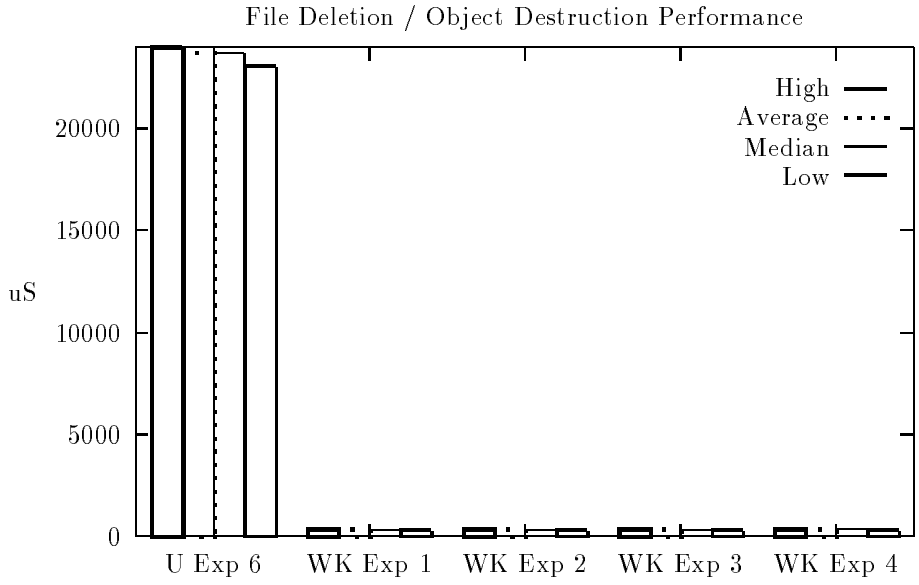


Figure 8.6: Comparison of File Deletion Time to Object Destruction Time

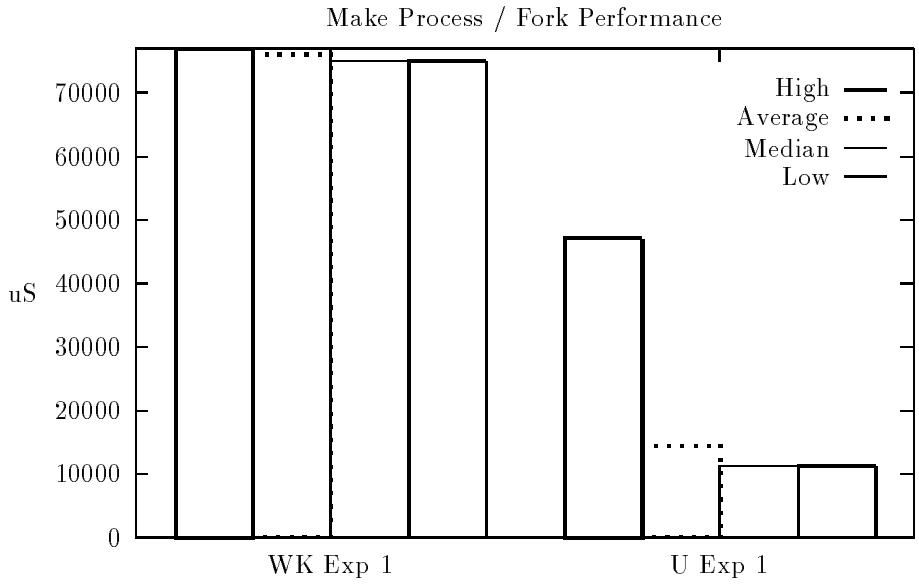


Figure 8.7: Comparison of Fork to Make Process Time

with differing contents (the equivalent of a *fork* followed by an *exec* under UNIX); and because UNIX returns as soon as it is clear that there are adequate resources for the new process to be created unlike the Walnut Kernel which returns as soon as the new process is ready to execute.

Chapter 9

User Level Programs

This chapter[1] describes programming techniques used for application programs and applications that have been written to operate under the Walnut Kernel. These applications have allowed programmers to explore the possibilities offered by a capability-based operating system and provided feedback to the operating system designers. This feedback has resulted in changes to the design of the kernel.

Four programs are described:

- Initproc - the initialization process
- Glui - a screen multiplexor
- Shell - a user shell
- Wyrm - an arcade style game

Initproc is responsible for deriving capabilities used by processes which manage access to devices. Shell and Glui form a user level interface which allows access to the functions of the kernel and to objects within the system. The game Wyrm is an example of a highly interactive application which demands fast response times from the system and is IO bound.

Program structures and data structures used in the applications are described in section 9.1. Section 9.3 describes enhancements to the Walnut Kernel's process

[1]The majority of the material contained in this chapter is reproduced from [CPW95].

structure to assist the implementation of shared libraries. Sections 9.4 to 9.7 discuss the application programs.

9.1 Structures

This section describes a number of common structures found in programs operating under the Walnut Kernel. It addresses both organization of programs and data structures.

9.1.1 Program Structures

Walnut Kernel programs are similar to programs which are implemented under GUIs in that both types of programs respond to external events. GUIs provide two constructs for handling events:

Message Loops are a loop which contains a call to a function that accepts an event from a queue of events, and then calls a function to handle the event.

Callback Functions are registered with the user interface and are invoked with a set of parameters when an event occurs. Callback functions are used to handle asynchronous events.

The Walnut Kernel supports constructs which perform similar tasks, but are implemented differently.

```
while true
begin
    wait(-1)
    receive(msg)
    server_function(msg)
end
```

Figure 9.1: Pseudocode for a Message Loop

The Walnut Kernel typically handles messages by using a **message loop** (see figure 9.1). This simple construct places the process (or subprocess) into a sleep

state until a message arrives, receives a message, handles the message, and returns the process to a sleeping state. As a process cannot sleep when there are messages waiting for it, the message loop can handle multiple messages without the need to test for the presence of a message before going to sleep.

Asynchronous events which would be handled with a call back function under a GUI are implemented through the use of subprocesses. A subprocess is a thread of control within a process to which a message can be specifically addressed. When a message arrives for a subprocess, the subprocess is made executable. Typically a message loop is used to receive and handle the message before putting the subprocess back to sleep.

9.1.2 Data Structures

Persistence, sharing and relocation shape the types of data structures in common use under the Walnut Kernel.

File oriented operations are typically performed on a stream of data, converting the contents of an input stream to an output stream. Persistent data structures do not require conversion to and from a secondary storage format, eliminating the stream orientation imposed by the file mechanism. In addition, programmers are able to perform random access operations on input and output data structures without the overheads that would be present on a stream oriented system. The absence of these constraints provides a new degree of freedom in the design of data structures.

A hash table is an example of a data structure that benefits from a persistent implementation. On a persistent system the hash table is stored in a directly usable form. This can be contrasted with a file oriented system which has the choice of extracting the data from the table and storing it in a linear form, or storing the table as a block of memory dumped to disk. The former requires either a complex transform on the data to recover it in the correct order for storage, or an additional data structure that keeps track of the order in which data should be stored. The latter approach requires the table to be read at the beginning of the program and written at the end of the program, introducing a significant IO overhead.

The easy sharing of data requires programmers to be aware of synchronization, access control, and locking issues. Currently systems programmers work in an environment where sharing considerations are important. Application programmers need to become aware of the issues and techniques for managing shared data. Provision must be made in shared data structures for collective access to the data structure. This may include choosing data structures that allow simultaneous access (circular buffers) or employ locking.

Programmers have a choice of loading an object at a fixed address or allowing the loading of an object at an arbitrary address. If an object is always located at a fixed address, pointers may be used within the object to refer to other parts of the data structure. This arrangement has the advantage of speeding references within an object. However, it causes a loss of flexibility and may restrict the sharing of objects. This is because programs will only be conveniently able to load one object at a time that occupies a set of points in the address space. Relocatable objects use index values to refer to parts of the object. This requires an addition operation before a dereference operation can be performed resulting in a potential loss of performance.

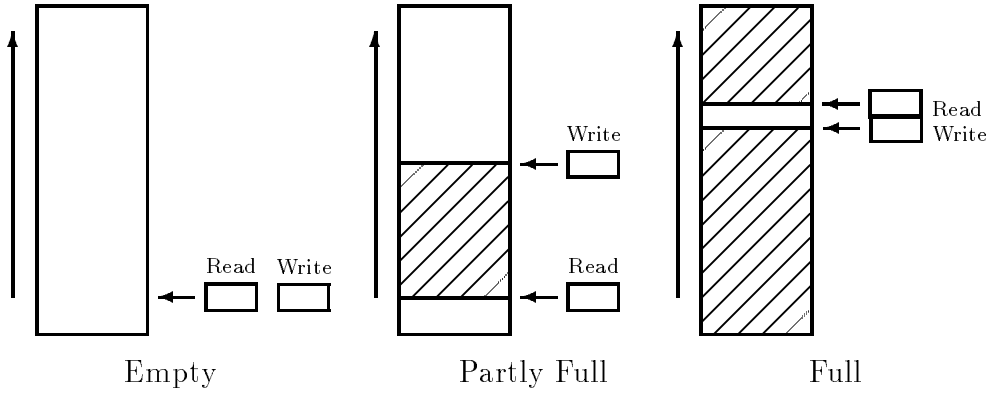


Figure 9.2: A Circular Buffer

Circular buffers (see figure 9.2) are used to transfer stream oriented information between processes. Two implementations are used:

- A minimal implementation is used by character mode devices such as serial ports and the keyboard to communicate with their manager processes.

- An optimized version is used for interprocess communication.

Both circular buffer implementations do not require locking, but ensure that data is correctly transfered from the sender to the receiver. They operate by giving the sender read/write access to the write-pointer, and read-only access to the read pointer. The receiver has read/write access to the read-pointer, and read-only access to the write pointer. This eliminates contention over updating the pointers. The data structure operates safely even if information relating to the position of the other pointer is old. The data structure has a minor inefficiency in that there is always a single wasted slot when the data structure is full.

In the more efficient implementation, both the sender and receiver have a private pointer known as the **tripwire**. The tripwire is set to point to either the value of the other pointer or the top of the buffer. Before sending or receiving an element from the circular buffer, the value of the pointer is compared against the tripwire to determine if there is the risk of overfilling the buffer or crossing the end of the buffer. This mechanism saves the cost of a comparison on most accesses to the buffer by converting the separate tests for overfilling and wrapping, from top to bottom of the buffer, into a single test. If the comparison indicates that either of the boundary conditions has been reached, further tests are carried out to determine which of the two conditions caused the problem, and the tripwire is set to a new position.

9.2 Legacy Code

A library has been constructed that emulates many of the functions found in the C *stdio* library. This library has two roles:

- It allows the reuse of a large quantity of existing C code, reducing development effort.

- It provides an environment that is familiar to a large range of programmers allowing them to use existing skills while learning about the features the Walnut Kernel environment offers them.

Under the emulation library files and streams are implemented using the same circular buffer code. Files gain no advantage from being implemented using circular

buffers; however, there is no performance penalty either. By choosing to implement the two mechanisms in the same way, code volume is reduced and code maintenance is simplified.

9.3 Shared Libraries

One of the major objectives of the Walnut Kernel was to encourage sharing of code and data. Section 9.1.2 discussed mechanisms which allow the sharing of data. This section describes existing mechanisms for the sharing of code.

The executable code may be either relocatable or non-relocatable. Relocatable code uses relative addressing to reference other parts of the library module. Non-relocatable code is linked using absolute addresses for all references. Relocatable code simplifies the implementation of shared libraries. In the absence of relocatable code it is necessary either to force modules to be always loaded at the same address or create multiple versions of a module at differing addresses.

Several classes of data[2] may be required by shared libraries: embedded data, shared data and data local to an instance[3] of the code. Embedded data consists of literal constants compiled into the code. Shared data is accessible to all invocations of a library and may be seen and modified by each instance. A variant on shared data is constant data shared by all instances but typically not modified. Data local to an instance of the code is private to that instance and is typically not accessible to other instances of the code.

The current implementation of the Walnut Kernel supports relocatable code. The password-capability model is well suited to supporting the majority of the classes of data. Embedded constants can be protected from alteration by not providing the write system-right on capabilities given to users. Sharing of data is achieved by loading capabilities. Providing mechanisms which support data local to an instance is less straight forward.

[2]This taxonomy of memory types is based on work conducted for the Monads project [Geh82].

[3]For the purposes of this discussion an instance of the code is created by loading a piece of library code into the address space of a process. If the code is multiply loaded into a process, there are said to be multiple instances of the code.

Section 12.1.2 describes a number of potential approaches to supporting shared libraries. The remainder of this section addresses current mechanisms and the modifications to the process structure required to assist their implementation.

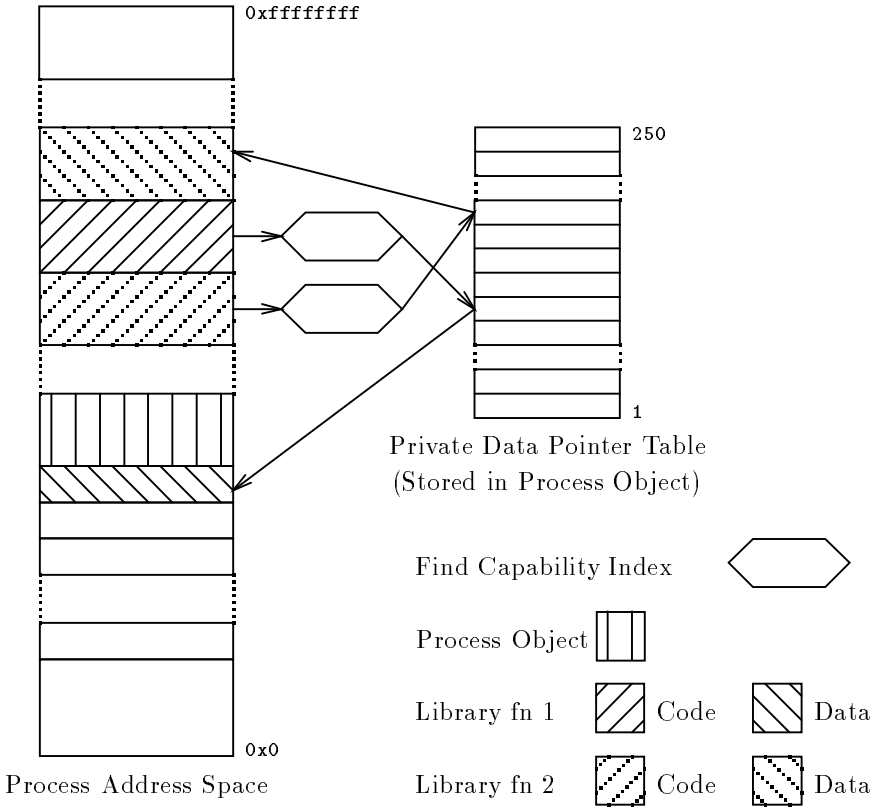


Figure 9.3: Implementation of Local Storage for Shared Library Code

To support data local to a process, a module must be able to create an object, load the object into the address space of the process and be able to locate the object to allow the module to read and write the module's local data. There are two varieties of solutions. The first variety involves structuring the address space of the process so that either modules are always loaded in the same place or providing a fixed relation between the location of the object used to store local data and the code object. Both these mechanisms place restrictions on the layout of memory. The second variety of solution is to store the address of the object containing local data in a location known to the module. This appears to be a catch-22 situation, as the module requires a private location to be able to store the address of its local data

object. This paradox can be avoided by providing a table stored - by convention - in the process object. The module has access to the address its code is loaded at and uses the address to index into the table. This provides a small private store readily accessible to a shared library module. The table is typically used to store pointers to a local data object. The mechanism is illustrated in figure 9.3.

A table indexed by the addresses of locations where objects can be loaded is relatively large. An enhancement which makes more efficient use of space is to use the Capability Index for the window in which the code is running. Capability indices range from 1 to 250, keeping the table down to a manageable size. Although it is possible to use the K_CAPID system call to find the index value for an address, it was considered too inefficient to use a system call for a potentially frequent operation. To assist in the rapid translation of addresses to Capability Indices, the process structure was altered to make the Address Map readable. This allowed a short segment of assembly code (see figure 9.4) to be used to rapidly translate addresses to index values.

9.4 Initproc

When a Walnut Kernel is booted, it generates an object known as the system object. This object contains all the memory pages occupied by kernel code, kernel data, and device driver interfaces and buffers. The initialization process derives capabilities from the system object used by the processes which manage devices. The **restrict** operation is then applied to the master capability. This operation removes rights associated with a capability without affecting the rights of the children of the capability. This eliminates a potential security hole associated with the existence of a capability allowing unfettered access to the kernel and device interfaces. After deriving the set of less powerful capabilities, Initproc notifies the scheduler that it is safe to schedule other processes, and sends messages to all manager processes containing the capabilities they require to access the devices they manage. Initproc completes its operation by entering a message loop and waiting for a message indicating that the system is to be reconfigured.

Initproc illustrates a number of features of programming under the Walnut Ker-

```
.globl _codecapindex

PROCHDADDRESS = 0x1000000
MAPADDRESS = 0x100f000

.text
.align 2, 144
/*
      addr = calling_address_magic();
      if (addr < PROCHDADDRESS)
         uqp = MAPADDRESS + ((addr >> 12) & 0xfff)
      else
         uqp = MAPADDRESS + ((addr >> 22) & 0xfff)
      return (*uqp);
*/
_codecapindex:
         movl (%esp), %eax
         pushl %ecx
         cmpl $PROCHDADDRESS, %eax
         jge bigwin
smallwin:
         movl $12, %ecx
         sarl %cl, %eax
         andl $0xfff, %eax
         jmp both
bigwin:
         movl $22, %ecx
         sarl %cl, %eax
         andl $0x3ff, %eax
both:
         movb MAPADDRESS(%eax), %eax
         andl $0xff, %eax
         popl %ecx
         ret
```

Figure 9.4: Find Capability Index for Executing Code

nel; however, the process is unique among Walnut Kernel processes in that it is restarted from a fixed address each time the kernel is booted. Apart from always starting Initproc from a fixed address, the kernel provides no special functions to support this code. Thus all of Initproc's code operates at the user level, requiring no special kernel support or privileges. All other processes resume their operations from the point at which they were stopped when the system was shutdown. Furthermore, as the system object is stored in volatile storage, the system object does not retain information about the capabilities applying to it over a reboot. The initialization process is responsible for remaking the capabilities used by the manager processes before allowing other processes to be scheduled.

The kernel scheduler monitors a word in the Wall. When the word becomes non-zero, the kernel scheduler allows the scheduling of any runnable process. Initproc derives a capability for the Wall from the system object. This capability is sent to the Wall manager and used by Initproc to notify the scheduler.

In addition to the easy sharing of data demonstrated by the above application, persistence is also exploited in Initproc. The derivative capabilities generated from the system object are stored in an array. When Initproc is restarted following a shutdown, it examines this array and generates derivative capabilities with the same name and rights as those found in the array before restarting the scheduler. This simplifies the design of the manager processes as the capabilities given to managers by Initproc appear to persist over the reboot. Holders of derivatives of capabilities distributed by Initproc will find that those capabilities no longer work.

9.5 Glui

Glui is the manager process for the screen and the keyboard. It provides several stream mode interfaces to the keyboard and screen. A series of keystrokes are used to switch between sessions. In addition, Glui supports a mechanism for giving direct access to the screen memory for a number of processes. Like Initproc, Glui functions using system calls available to all processes.

When the Walnut Kernel is booted, Initproc sends a message with a capability for the resources managed by each manager process. On receipt of the messages

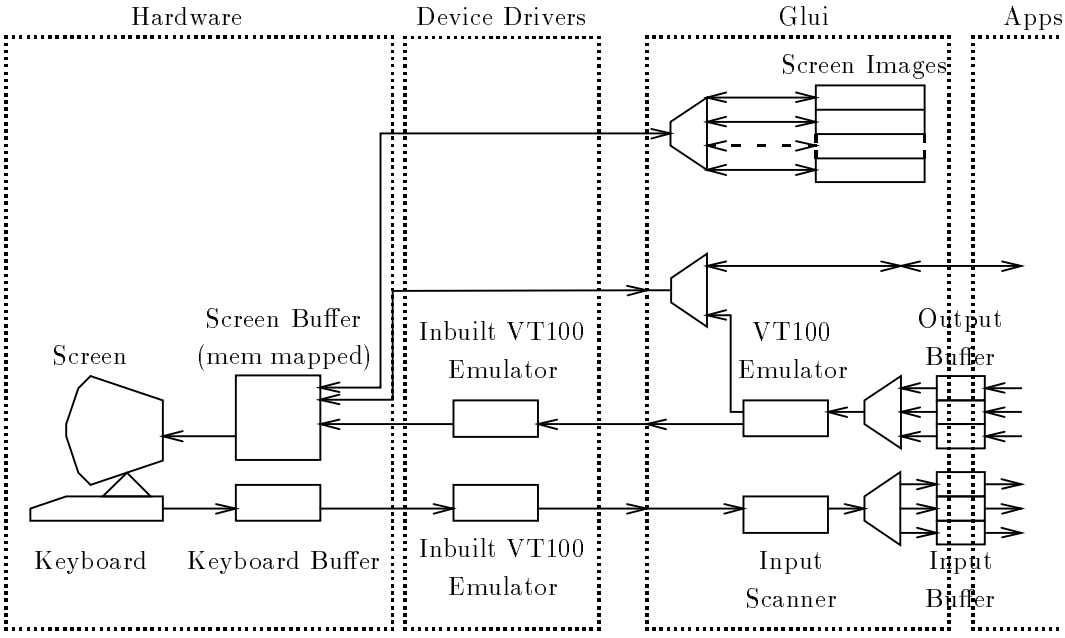


Figure 9.5: Keyboard and Screen IO

containing capabilities for the keyboard, the screen, the VT100 emulator built into the kernel, and the Wall, Glui creates 10 virtual screens and derives a capability which allows messages to be sent to Glui. This capability is then placed on the Wall.

The screen and keyboard IO architecture of the Walnut Kernel is illustrated in figure 9.5. To provide terminal multiplexing facilities, Glui intercepts all keyboard input scanning for control sequences. If no control sequences are found, the keyboard input is placed in the input buffer for the application currently being displayed on the screen. The output buffer of the current application is polled periodically. If new information is found in the buffer, it is passed to the VT100 emulator code built into Glui. This emulator writes its output directly to the memory mapped screen buffer. To move the cursor and sound the bell, Glui passes control codes to the VT100 emulator built into the device drivers.

To change the display to another application, Glui stops accepting input from the current client program. The current contents of the screen are copied to a buffer associated with the current application. This buffer is located within Glui and is not made accessible to other programs. The buffer corresponding to the new application

is copied to the screen, and the output buffer of the new application is read to update the screen. Keyboard output is directed to the input buffer of the new application.

Glui supports two types of output services:

- a VT100 emulator
- the hardware screen buffer

When a process requires IO through the VT100 emulator and keyboard, it sends a message to Glui using the capability on the Wall. If there is a virtual screen available, Glui sends a message back which contains the capability for a keyboard buffer and a screen buffer. These buffers use the circular buffer protocol discussed in section 9.1.2. The process requesting the screen may send data containing VT100 screen control sequences via the output stream. Input is received via the input stream.

When direct access to the hardware screen buffer is requested, the process must supply for itself a capability that allows the process to be frozen. If this capability is not provided, or does not allow Glui to send the freeze message, the request will be rejected. If a suitable valid capability is supplied, a capability without SRMULTILOAD right and with a password 2 equivalent to the requesting process's serial number is returned to the requesting process. When loaded by the the requesting process, this capability allows direct access to the screen buffer; however, this capability cannot be loaded by any other process.

Protected freeze and thaw are used on the processes granted direct access to the memory mapped screen buffer. This prevents other processes from thawing a process with a usable capability for writing to the screen. Glui is able to ensure that only one process writes to the screen at a time, preventing corruption of the screen's contents.

Both the SRMULTILOAD right and the protected versions of freeze and thaw were introduced to enable Glui to allow controlled direct access to the hardware screen buffer. Other solutions were considered, including locking processes [AW85] and schemes for the rapid revocation of capabilities.

Under the Walnut Kernel, a process is locked when it is created with a 63-bit

lockword. This lockword is XORed with each 'alter' capability[4] before the capability is used by the kernel. The process can only use capabilities which have been XORed with the lockword, and then be passed to the process. This prevents a locked process from communicating with other processes without the assistance of a party who knows the lockword value. This mechanism was considered, but locking severely curtailed the ability of the client program to communicate.

Although a number of rapid revocation schemes were considered, the generalization of these schemes to a multiprocessor environment either resulted in a mechanism insufficiently responsive, or required an unacceptably high overhead to support a relatively infrequent operation.

9.6 Shell

Shell is a command interpreter. It provides mechanisms for managing objects, organizing 'files' generated through the stdio emulation code and launching programs. Shell has detailed information relating to the structure of a process which follows the conventions adopted for the Walnut Kernel.

When Shell is first started it sends a message to Glui requesting a terminal emulator output buffer and a keyboard input buffer. On receipt of these capabilities, it presents the user with a prompt and awaits further instructions. Users can run processes in two modes:

- Yielding the screen to the new process. The input buffer and output buffer used by Shell are given to the new process for its use until the new process terminates.

- Creating a new screen for the new process. The shell requests a new set of buffers from Glui which are given to the new process.

The two modes differ in several respects. When the new process is to inherit the screen from Shell, the buffers are made available to the new process and the shell

[4] A non-alter capability does not possess write rights and cannot be used to transfer information to another process. Alter capabilities can be used to transfer information.

goes into a loop which polls the new process's status. Shell ignores the contents of the buffers and does not take any command input until it detects that the new process has ceased to function. Shell then resumes using the buffers and accepting commands. When the screen is not inherited from Shell, a set of buffers is requested from Glui and made available to the new process. Shell continues to interpret the input from the keyboard and remains active on the screen that it is currently connected to.

Two mechanisms were introduced into the Walnut Kernel to allow processes to determine the state of another process:

Cooee Messages are sent to a process and results in a **Cooee reply** message being sent to a capability specified in the cooee message. The Cooee reply message is automatically generated by the kernel and contains a field indicating whether the process is running, frozen, sleeping or dead.

Peek System Call returns a value which indicates whether the process is running, frozen, sleeping or dead.

The Cooee message was introduced first; however, polling processes to determine their state proved to be useful and popular, so the more efficient peek mechanism was provided. The peek mechanism has the advantage of a significantly lower overhead as it requires only a single system call and the message passing mechanism is avoided.

A process object conforming to the Walnut Kernel conventions contains:

Startup Code Area (optional) This area may contain a small amount of code used in starting a process.

File Descriptor Table (mandatory) This area contains the file descriptors for use by the process. Note: The first 3 elements of the File Descriptor Table are mandatory to allow for standard output, standard input and standard error. The entries in this table are used by the Unix emulation library.

Private Data Pointer Table (mandatory) This area contains pointers to private data. The table is indexed by the capability index of the executing code and is used to locate data used by the executing code.

Default Heap (optional) The default location for the creation of the heap.

Default Stack (optional) The default location for the creation of the stack.

To start a process, the shell takes a code object and a data object for the program to run in the new process. The data object is duplicated. A new process is made by invoking the kernel. The capabilities for the code object and the duplicate data object are passed to the kernel as autoload capabilities[5], the stack pointer and program counter are set and the wakeup time for the new process is set to forever. After the new process is created, Shell, modifies the pages of the object loaded into the shell's address space. Shell writes into the file descriptor table the capabilities for the new process's standard input, output and error, and any other file descriptors that are required. The command line arguments and a capability for the object containing the process's environment strings are written into the heap space. A message is sent to the process to wake the process up.

The method used to create processes allows multiple copies of a program to be run simultaneously. The scheme is economical of both disk space and memory space as it shares a single image of the code. The data is duplicated to prevent multiple copies of a program interfering with each other.

9.7 Wyrm

Wyrm[6] is an arcade style game inspired by the games nibbles[Cor90] and worm[Toy91]. Apart from its frivolous value, Wyrm has been used to test the responsiveness of the interface and a number of IO mechanisms.

The current version of Wyrm makes use of the Unix emulation stream IO code to communicate via standard IO with Glui which draws the parts of the game on the screen. This version is highly responsive and shows that the two layers of software

[5] Autoload capabilities are automatically loaded into the address space of a process when the process is created.

[6] A wyrm is a mythical creature of great power. The game was sarcastically named wyrm because of its lack of speed. After tracing a number of implementation problems in the stream IO code Wyrm now proudly lives up to its name.

provided by the existing IO structure are sufficiently quick for highly interactive applications.

Early in the development of the Unix emulation libraries a misplaced *fflush* had caused us to believe that the system performance was inadequate. At that time, a version of Wyrm which made direct use of the screen was written in an attempt to determine where the bottleneck lay. This resulted in changes in the design of the kernel and Glui to correctly support the sharing of memory mapped buffers.

Chapter 10

Security

The Walnut Kernel differs from the Password-Capability System in a number of respects. This chapter outlines the effect of these differences on the security of the Walnut Kernel. The Password-Capability System required that objects be continuous and that there be sufficient storage to contain the object when the object was created. The Walnut Kernel has paged objects which may have unallocated pages within the object. Other significant differences include the introduction of the **restrict** operation by the Walnut Kernel, serial numbers having a physical meaning, capabilities with non-random passwords, the SRMULTILOAD right and the **protected freeze** and **thaw** operations.

10.1 Objects

The Walnut Kernel uses three parameters to describe the storage and address space requirements of an object. The Password-Capability System used a single parameter to describe these features of an object. The separation of storage requirements of an object from the size of the address space used by the object was aimed at overcoming the limitations imposed by having a page sized protection granularity. It also provided an opportunity to introduce a new degree of freedom into the Walnut Kernel not found in the Password-Capability System. Figure 10.1 compares the structures of the two systems' objects.

The decoupling of the address space requirements from the storage space re-

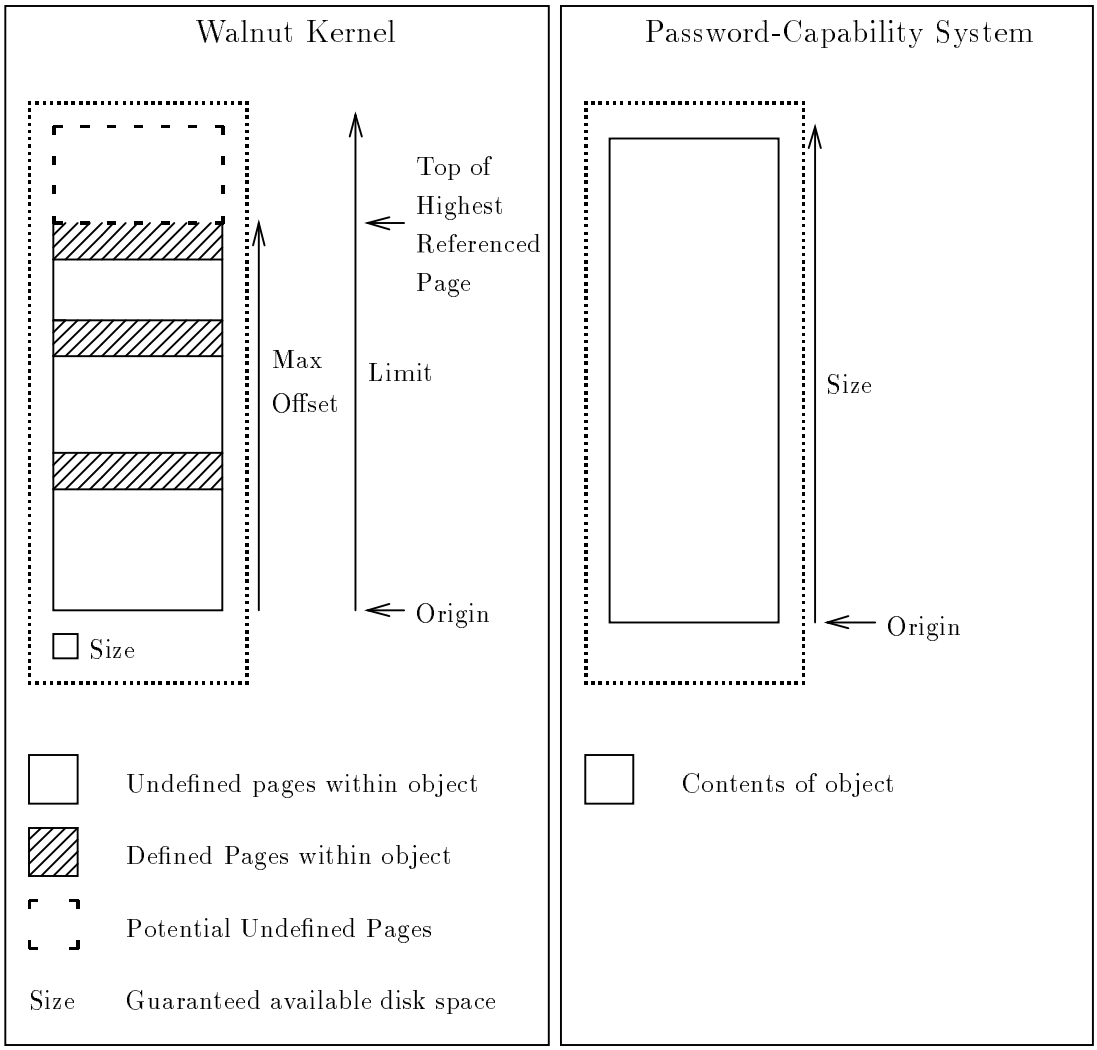


Figure 10.1: Comparison of Walnut Kernel and Password-Capability System Objects

quirements increases the flexibility of the object mechanism. In the Walnut Kernel, several uses have been proposed that make use of this flexibility. These applications include the construction of single-object processes and the storage of source code and its derived object code in the same object. Compiler designers tend to use a large address space and populate it with widely separated code and data segments. Compilers using this approach on the Password-Capability System would have committed a large amount of system disk resources. In contrast the Walnut Kernel requires fewer resources.

The second suggested application places code into the same object as the compiled output. This mechanism ensures that source code is available to maintenance programmers. The wide separation of the starting address of the source code from the compiled object code eliminates the need to relocate code on code growth. This simplifies the task of locating the start of the sections of an object, and allows the object to be easily partitioned into source and code segments using the capability mechanism.

The mechanism of assigning the pages of an object when the page was first accessed introduced a secret channel. The channel resulted from the changing of the state of an object when a read operation was carried out on memory accessed through a read-only capability. To transfer information through this channel the following pre-conditions must be fulfilled:

- All the pages of the volume must be allocated to objects on the volume
- There must be a known number of unassigned pages in the object through which the information is to be conveyed and the location of unassigned pages must be known.

The transmitter of the message sends it by reading from previously unassigned pages. This results in the pages being assigned. After the assignment has taken place, the receiver of the message determines the number of pages left by reading from a collection of unassigned pages[1] until a fault results from attempting to allocate a

[1]The collection of pages used by the receiver must be disjoint from the set used by the transmitter

page. The address-fault exception handler of receiving process is invoked, and the number of pages the receiver has managed to access corresponds to the message.

The existence of a new secret channel has been demonstrated. However, the transmission mechanism can be easily frustrated by either freeing space on the volume, restricting information relating to the assignment of pages in shared objects, or ensuring that read-only capabilities are assigned only to sections of objects which are completely assigned.

10.2 Restrict

The security analysis of the Password-Capability System was based on a model which required child capabilities to be no more powerful than their parents. The introduction of the **restrict** operation allows parent capabilities to be made less powerful than their children. The Walnut Kernel modified the requirement on child capabilities to apply only at the instant of creation.

The initial motivation for restrict was to provide a mechanism which allowed the kernel memory object to be made available, safely, to the initialization process. When the Walnut Kernel is started an object is created which covers the kernel program and data areas. The initialization process derives smaller views from the kernel memory object, which contain buffers used to communicate with hardware devices, and kernel variables which may be used to send information to processes or allow possessors of the capability to perform restricted functions[2]. Possession of the master capability for the kernel memory object allows the holder complete control of the system. Early versions of the Walnut Kernel used a capability with a well known value for the kernel memory object[3]. The restrict call was introduced to allow the initialization process to remove all the rights of the kernel object's master capability after the smaller views were derived.

[2]This mechanism has been proposed as a method of allowing the kernel to be notified that the system should be shutdown.

[3]The current system passes the value of a randomly selected master capability to the initialization process. The restrict call is used at the completion of derivation as an additional security measure.

The restrict operation reduces the rights of a capability by performing a bitwise-and of the rights mask supplied with the rights bitmap of the capability. The restrict operation will only operate on capabilities which have suicide right.

To show that the Walnut Kernel retains the security properties of the Password-Capability System, it is necessary to show that the restrict operation introduces no behaviors that cannot be achieved under the Password-Capability System. The restrict operation places a system enforced limitation on the usage of a capability. The limitation is equivalent to a program on the Password-Capability System voluntarily foregoing the use of some of the rights conveyed by a capability. Thus, the Walnut Kernel is no more subject to rights amplification than the Password-Capability System.

Appendix B contains a formal description of the effects of the restrict operation.

10.3 Serial Numbers

The Walnut Kernel uses the less significant bits of the serial number to locate the header page of the object. The remaining bits of the serial number are randomly allocated (see figure 10.2). When an object is referenced, the block number of the header page is extracted by performing a bitwise-and of the Disk-Block-Mask and the presented serial number. The header page is accessed and the presented serial number is compared with the serial number stored in the header page. If they match, the operation which made reference to the object is allowed to continue. If there is no match, an error is returned indicating that the capability is invalid. This differs from the mechanism in the Password-Capability System where serial numbers of objects were allocated randomly, and a table was used to convert the serial number to a reference to the physical representation of an object.

This change has the potential to reduce the security of the system by reducing the size of the name space that an individual must search to find a valid capability. In practice the mechanism has a minimal effect on system security. It can be shown that the impact of this change is equivalent, in the worst case, to reducing the serial number's length by 1 bit. The proof follows:

The random bits of the serial number are unpredictable hence those bits do not

$$S = B + (R\&\overline{M})$$

B Block Number of Header Page

M Disk Block Mask

R Random Number

S Serial Number

Figure 10.2: Components of the Walnut Kernel Serial Number

reduce the size of the space which must be searched.

The user has limited knowledge of the state of the physical device. The user can only determine the header blocks which have been allocated to capabilities the user already knows. This is the same level of knowledge as found in the password capability system. Furthermore, as the system operates, objects are created, destroyed, enlarged and shrunk. This results in a continually changing set of candidate header blocks. Selection from the changing random pool results in a random series of potential header blocks.

Unless the disk holds exactly a power of two blocks, the most significant bit of the block number will not be biased towards zero. The worst case would occur when there is only a single block number with a serial number in which the top bit is set. This would result in reducing the search space by a bit.

Assuming the worst case, the serial number is 30 bits in length[4]. Although the serial number of an object was never regarded as a secret, a 2^{30} item search space for serial numbers is sufficiently large to make random probing for capabilities not viable.

10.4 Non-Random Passwords

The Walnut Kernel introduced a mechanism for creating capabilities with passwords specified by the process performing the **derive** operation. This mechanism was introduced to allow the initialization process to create capabilities derived from the

[4]The top bit of the serial number is used to mark alter capabilities

kernel memory object with known passwords. The kernel memory object has the unique feature of retaining no state over a system reboot. All other objects reside on a physical medium allowing the state of the object to remain in existence until the object is destroyed. Accordingly, all the capabilities derived from the kernel memory object are forgotten whenever the system reboots. Without the ability to generate capabilities with known passwords, all processes using capabilities derived from the kernel memory object would, after a reboot, have to handle the loss of access to the view provided by that capability. The initialization process regenerates the capabilities for the kernel memory object after each reboot before allowing other processes to restart. Regenerating the capabilities simplifies the processes which interface with the kernel memory object.

The ability to create capabilities with non-random passwords is essential to allow capabilities without the SRMULTILOAD right to perform a useful role.

The creation of capabilities with known passwords has a minimal effect on the security of the system. As the random password mechanism remains, the user is still able to create capabilities which have a full range of password values providing the same level of security as in the Password-Capability system. In addition, the non-random mechanism can be used, with no impact on system security, in the two following roles:

- A capability can be made available to a wide range of users by creating a capability with a known name. Using a known name is equivalent to widely advertising a capability with a random password.

- Deleted capabilities can be replaced by a possessor of a sufficiently powerful capability. Accordingly those who knew the original name of the capability can make use of the new capability created with the same passwords, entailing no further spread of information

10.5 SRMULTILOAD Right

The SRMULTILOAD right was introduced to allow capabilities to be created which could only be used by a specified process. The screen manager uses this mechanism.

The screen manager acts as a screen multiplexer supporting several text screens. The user can switch between screens by using a hot key sequence.

To speed up access to the screen and allow greater control over the screen, the manager can grant direct access to the screen's video buffer to a process. However, the screen manager needs to be able to ensure that only one process can exercise write access to the screen buffer at a time. When the hot key sequence is pressed, the screen manager uses a protected freeze to prevent the process with a capability to the screen buffer from executing, and switches control of the screen to a new process. In this example, the screen buffer capabilities given out by the screen manager do not have SRMULTILOAD right; so only the designated process can use them. Other processes cannot use the capability so if the capability is given to a third party the screen's contents cannot be altered through the use of that capability.

Capabilities without SRMULTILOAD right can only be used by processes whose serial number is equal to the value of password 2 of the capability.

This mechanism allows only a decrease in the number of potential users of a capability thus allowing tighter control over the use of capabilities. As it restricts the potential domain of use of a capability it does not decrease the security of the system.

10.6 Protected Freeze and Thaw

The **protected freeze** and **thaw** mechanism is an enhancement of the freeze and thaw mechanism that allows a process to ensure that a process under its control cannot awake unexpectedly through the action of another process.

The screen manger example (section 10.5) used for the SRMULTILOAD right also employed a protected freeze and protected thaw. The motivation for the introduction of this new mechanism was that the original freeze and thaw mechanism could not prevent a third party process from waking up a process holding a capability for a screen buffer and hence altering the contents of the screen.

The protected freeze mechanism and thaw mechanism uses a magic number to prevent a process from being thawed unless all the processes which used a protected freeze on a process have thawed the process. The magic numbers are XORed together

and a process is only runnable when the number of protected freezes equals the number of protected thaws and the product of the XOR operations is zero.

This mechanism provides an enhancement on an existing mechanism. The basic properties of the mechanism do not affect the security model of the Walnut Kernel. The mechanism has enhanced the security of the system by allowing a process to exert control over the execution of another process without the risk of a third process overriding the inhibition of execution.

Chapter 11

Proposed Hardware

This chapter describes a proposal for a novel hardware environment on which the Walnut Kernel may operate. The proposal presented here was originally jointly developed by Dr Ronald Pose of the Department of Computer Science at Monash University and myself in 1993. A paper [CP94] outlining the design of a node of the multiprocessor and a mechanism for avoiding the problems associated with a global system clock was presented to the Seventeenth Annual Computer Science Conference. This paper is reproduced in appendix C. Further work based on this proposal has since been undertaken by Dr Pose and a number of his postgraduate and honors students [PFR94, FPR95].

11.1 Design Goals

The Secure RISC Multiprocessor Project undertook to design a scalable general purpose multiprocessor. The architecture was required to support a wide range of sizes varying through: single processor workstations, multiprocessor workstations, medium size multiprocessors and large clustered multiprocessors.

In addition the system was required to be built of modular components, use a passive backplane for interconnecting processors, provide high performance by minimizing the potential for performance limiting bottlenecks in the bus structure, be sufficiently flexible to support a variety of algorithms - rather than being tuned to a specific class of algorithms - and provide mechanisms to support fault tolerance.

A key requirement was to allow the number of processors to be increased gradually when demand for processor power was required and budgets allowed. Large commercial multiprocessors tend to scale incrementally up to the size of the interconnection network. Scaling beyond this size typically requires replacing the interconnection network with a larger network. This can make upgrading prohibitively expensive.

To achieve the goal of gradual scaling, we proposed an architecture where the switching network was distributed evenly across the nodes of the system. We also determined that the speed of processors and memory must be allowed to vary from node to node. This decision allowed older, less fast, nodes to be retained in a useful role in a system which had acquired more recent faster processor and memory units while still exploiting the new units to their maximum potential.

11.2 Architecture

The design goals eliminated a number of popular multiprocessor organizations. Neither hypercube nor tree architectures were suitable as they lacked adequate path redundancy for fault tolerance and they were unable to support well a wide variety of algorithms. Dr Pose put forward a series of bus topologies with varying degrees of interconnection between buses. These designs were mesh-based, providing redundant paths and allowing for the possibility of minimising bottlenecks by routing around network 'hotspots'. We converted the proposed bus interconnection schemes into a regular[1] implementable form. The resulting 4-way and 6-way interconnection designs are illustrated in figure 11.1. Furthermore, we observed that the patterns could be converted into *cylindrical* and *spherical* forms by *folding* the mesh in half and connecting busses at the edges of the mesh.

Figure 11.2 provides the conventional representation of the processor interconnection topology. The diagram shows the direct connections between processors. The 4-way interconnection pattern allows a processor to directly communicate with 6 other processors, and the 6-way interconnection pattern allows a processor to

[1] A structure of repeated subunits

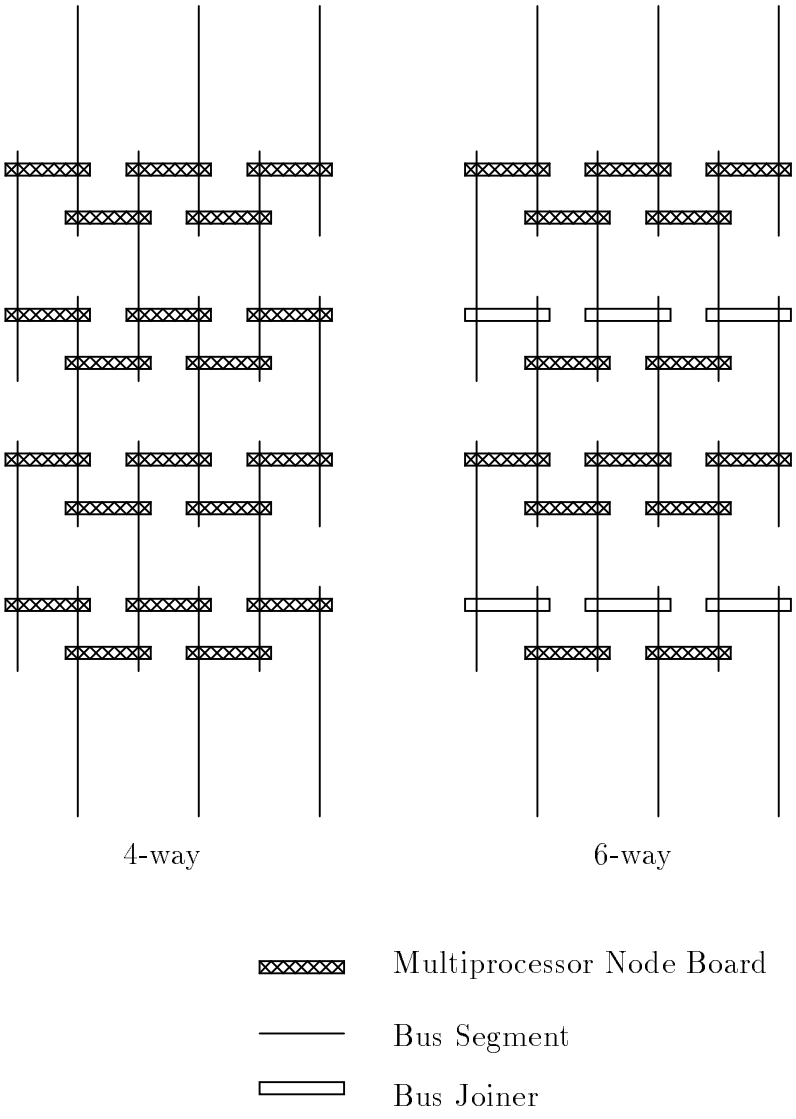


Figure 11.1: Bus Structures

communicate with 10 other processors.

Conventional clock distribution mechanisms were unlikely to be sufficiently scalable to be used on an architecture with widely variable topologies and bus lengths[2]. To meet the demands posed by this architecture we proposed a system built from a single type of node. Each node had its own clock which it exported to the other nodes on its bus segment when communicating.

11.3 Node Design

The Multiprocessor Node board (Figure 11.3) supports a combination of processor or memory modules on the M-P bus (Memory - Processor Bus). These modules are to be constructed on daughter boards which are plugged into the Multiprocessor Node board. A single Multiprocessor Node board behaves as a classic SMP machine. Using the 2 external ports it is possible to connect to an external network of multiprocessor node boards using a passive backplane.

Each Multiprocessor Node Board has a local clock pulse generator. This is used to provide clock signals to the processor and memory daughter boards, the control logic, and the arbiters. This clock is also gated out through the ports to clock the external bus when the port becomes a bus master.

The requirement for a global clock is eliminated by using the FIFOs to decouple the local clocks from the clock found on the external bus.

11.3.1 Functional Description

The Multiprocessor Node Board consists of 4 major functional blocks connected by a state machine (the control logic). The blocks are:

- M-P Bus
- Bus Switching Unit

[2] A technique known as 'Salphasic Clock Distribution' [Chi90] can be used to distribute a synchronous clock signal with the required accuracy. However, a synchronous design does not allow processors to operate at arbitrary speeds, reducing debugging flexibility and negatively affecting the cost of system expansion

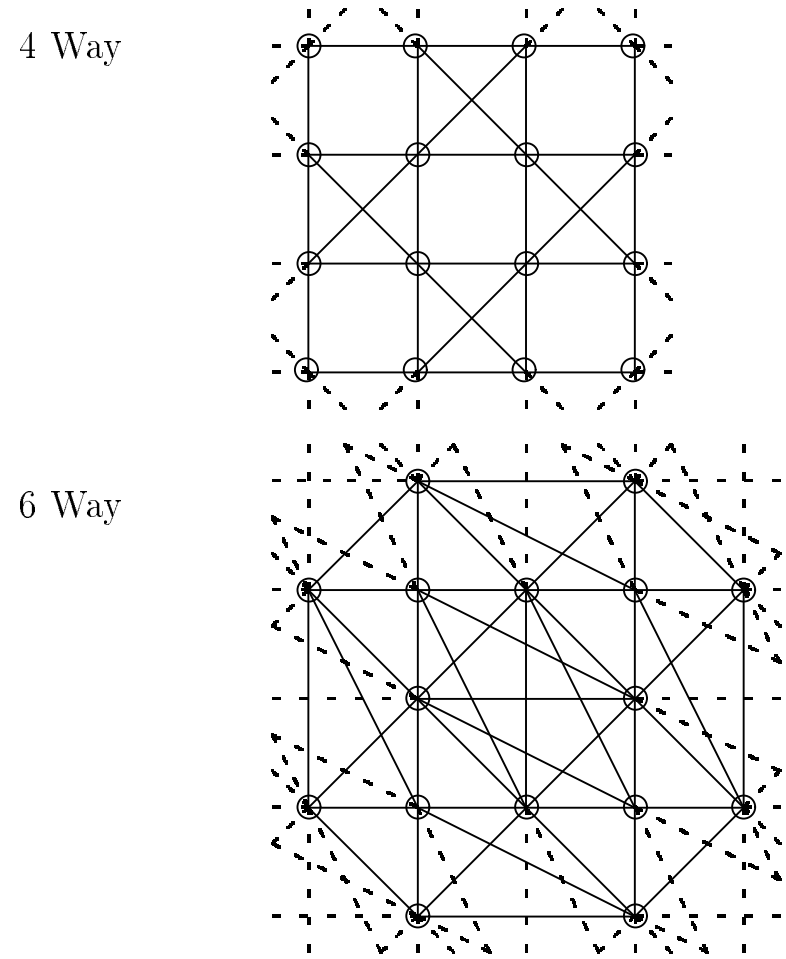


Figure 11.2: Topology of Processor Interconnections

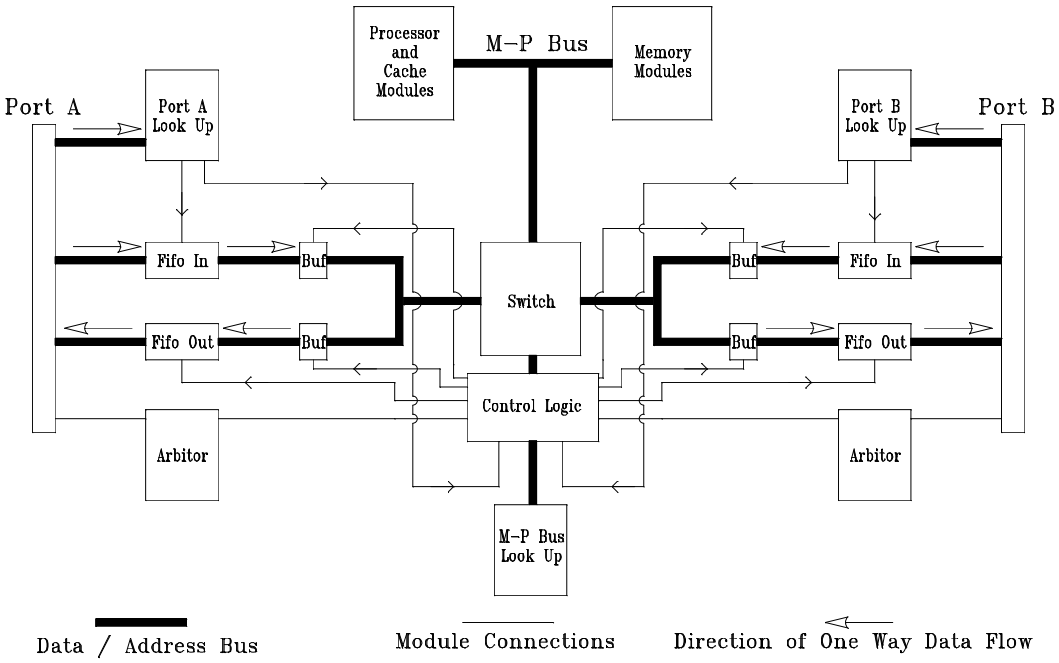


Figure 11.3: Block Diagram of Multiprocessor Node

- 2 Port Interface Units

M-P Bus

The Memory-Processor bus (M-P bus) is a data, address, and control signal bus. The data and address paths are 64 bits wide, with the data and address signals multiplexed onto the bus. This bus runs using a split bus protocol provided by the processors[MIP91] on the daughter boards plugged into the Multiprocessor Node Board.

The processor units and memory units on the M-P bus form a classical shared memory, SMP machine.

Bus Switching Unit

To help provide off-board communications the switching unit provides 4 operational states:

- Port A connect Port B
- M-P Bus connect Port A

- M-P Bus connect Port B
- No Connection

Port Interface Units

Each port has a port interface unit which performs the 2 functions of transmitting data onto a bus and receiving data from the bus.

To receive data this unit recognizes relevant information on the bus and accepts it into the input FIFO, otherwise bus traffic is ignored.

To transmit data the port interface unit arbitrates for the bus, and then outputs data from the output FIFO.

11.3.2 Operational Description

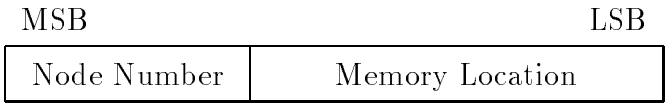
All addresses in the system are partitioned into 2 regions. The most significant bits of the address determine which Multiprocessor Node Board is to be accessed, and the least significant bits determine the address of the memory location on the Multiprocessor Node Board (see Figure 11.4). Two Multiprocessor Node board numbers are reserved: node board number zero always refers to memory local to the node board, and the maximum node board number refers to hardware control memory local to the node board.

The Node Number is used to index into a routing look-up table held in static RAM, which is decoded to determine where the memory location can be found.

There are three types of access available to the processor:

- Local Memory - Memory is addressed directly over the M-P bus.
- Remote Memory - *Discussed in Section 11.3.2*
- Hardware Control Memory - The bus ports are isolated and the routing (look-up) tables are modified by the processors.

In addition a Memory to Memory DMA transfer facility is available to facilitate page sized transfers.



Node Numbers:

$0 \ldots 0$	:	Local Memory
$f \ldots f$	:	Hardware Control
$x \ldots x$	:	Off Board Memory

Figure 11.4: Partitioning of Addresses

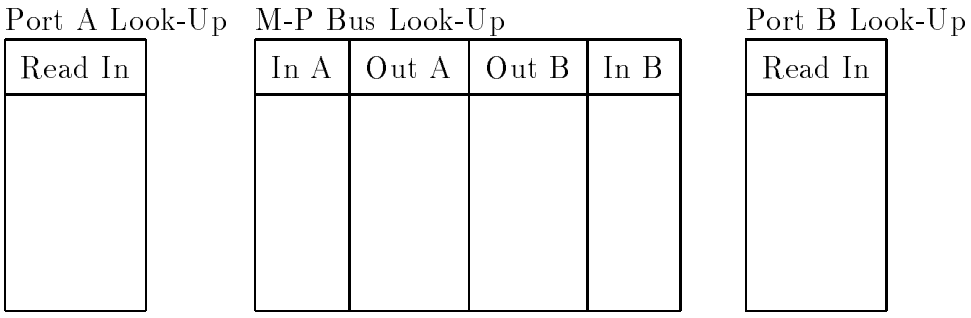


Figure 11.5: Contents of Look-Up Tables

Routing

This section illustrates the operation of routing data between memory and processor by following the path of a memory access

Transfers between nodes employ a packet structure. A packet comprises a header, a body containing the data and a packet check sum. The header contains the source and destination addresses, the packet size and the packet type. Packet types include read, write and an indication of whether the destination is a processor or memory. Packets are constructed and interpreted by control logic in the multiprocessor node board.

Local memory operations use the intrinsic addressing mechanism of the processor.

Memory accesses are routed through the network in a manner similar to a packet based store-and-forward network.

When a processor utters an off-board address, the high order bits of the address are used to index the M-P Bus look-up table. The look-up table contains bits which indicate which port should be used to attempt the access (Figure 11.5).

If the output FIFO on the required port is below the high water mark (the point at which it is guaranteed that the largest permissible packet will fit in the FIFO) and there is no traffic currently passing through the switch, then the switch is connected to the appropriate port. A packet header is constructed and transferred to the FIFO. Data is transferred to the output FIFO. A check sum is added to the FIFO. If the conditions are not met the processor should reattempt the operation later.

When there is data in the output FIFO and the bus to which the port connects is idle, an attempt is made to arbitrate for the appropriate bus (The arbiter is discussed in Section 11.3.4). When the port becomes the bus master the packet is broadcast onto the bus.

The high order bits of the destination address of the packet on the bus are used to index into the port look-up tables of all ports attached to the bus. If the port's 'Read In' bit is set and the input FIFO is below the high water mark, then the data on the bus is read into the FIFO, otherwise the data is ignored.

The node number of the destination address in the header of the first packet in the FIFO is used to index the M-P Bus look-up table. If the 'In' bit is set, the switch allows the contents of the packet to be directed to the memory on the M-P Bus; otherwise the switch is set to permit the flow of data from the input FIFO to the opposite output FIFO.

11.3.3 Design Features

The M-P bus and switched external memory packet transfer allows better utilization of processor memory resources. Both the external and M-P buses may be loaded to the level providing optimal utilization of the bus capacity.

The design introduces a memory hierarchy based on the number of hops between nodes. This feature introduces a new degree of flexibility in the management of

both memory pages and processes. The throughput of a process is maximized by relocation of the process and/or its data to minimize the memory access time. The optimization of overall system performance is complicated by memory's being shared by multiple processors. Peak performance is achieved by balancing processor load, memory load, and process average access time [BFS89].

By employing FIFOs on each port, the risk of being unable to accept data on a port due to traffic on the M-P bus to the other bus's port is reduced. However, this design decision has the cost of adding latency to every transfer through a multiprocessor node. The presence of FIFOs on each port is especially valuable where large packet transfers are expected as it effectively doubles the depth of the FIFOs for flow through traffic, hence reducing the risk that a packet will not be accepted because a FIFO is above the high water mark.

11.3.4 Arbitration

The project has considered many arbiter designs. The designs were evaluated against the following required properties:

- Fairness - Each node must have a similar probability of becoming bus master as the starvation of a node would prevent the delivery of data.
- Guaranteed Result - A bus master is selected every time an attempt is made.
- Varying Asynchronous Clocks - Arbiters are synchronous with respect to their local clock.

The mechanism proposed in the paper used a priority based arbiter to resolve bus master conflicts. Priorities were rotated after each arbitration to ensure long-term fairness. The implementation employed a set of bus-request lines and a set of acknowledge lines to provide a handshake. This ensured that the priority-based phase was not undertaken until after all nodes on the bus had acknowledged that they had recorded the request state of all other nodes. The recording of the request state of other nodes was necessary to reduce the effects of potential meta-stabilities in the components of the arbiter on the result. The implementation was deficient

in that it reduced the arbitration speed to a multiple of the speed of the clock on the slowest board on a bus. A variant on this design was proposed which used a clock which was independent of the node's master clock to supply the arbiters' requirements. The independent arbitration clock could operate at a high speed and could be run at a uniform rate across the boards resulting in an improvement in the speed of arbitration and a simplification of the arbitration circuitry.

Dr Pose currently has students working on alternative solutions to be implemented in VLSI. Work undertaken by Ted Kehl of the Department of Computer Science and Engineering at the University of Washington in the areas of self tuning of VLSI circuits and a mutual exclusion element which restricts the spread of metastable voltages [KB] offers the opportunity to produce a simple arbiter with the required properties.

Chapter 12

Continuing & Future Work

This chapter outlines likely future developments in the Monash Secure RISC Multiprocessor project for the system software and the hardware. Section 12.1 covers software and section 12.2 covers hardware developments.

12.1 Software

The Walnut Kernel provides an environment which supports persistent shared memory. This section outlines further work on the kernel itself and on a number of projects which exploit its features. The section also details the advantages of using a capability based operating system for these tasks and problems likely to be encountered by implementors of these projects.

12.1.1 Kernel

Development of rent and charging code have both been postponed in the current version as their presence was not required during the initial development phase. It is planned to implement these monetary functions and disk reconstruction code in the near future.

Although the current kernel has functions which allow the manipulation of money, it does not contain code to implement charging by kernel operations and rent collection. Provision has been made for the implementation of these features.

Charging for kernel services is intended to be performed by debiting a process a fixed amount of money per service when the service is requested. This allows the budget required for the needs of a process to be easily calculated.

The **rent collection** mechanism is intended to run periodically. It will examine each object mounted on a volume and debit the appropriate amount of rent. Typically the rent collector will run when the system load is light. As the rent collector may visit objects at irregular intervals[1], it is necessary to store a time stamp in each object to determine when rent was last collected and the amount currently due.

At present the Walnut Kernel provides no mechanism for the recovery of corrupted volumes. Provision has been made for reconstruction software to be incorporated. It is expected that the reconstruction software would consult the bitmap which describes the allocation of pages on the volume and scan the volume for pages containing magic numbers indicating that the page contains a header page. By comparing this information it should be possible to confirm that a volume has not been corrupted when the volume is mounted and failing that restore at least part of the lost data.

12.1.2 User Code

User Interface

The Walnut Kernel currently supports a text based interface which provides screen multiplexing - the interface provides a number of virtual screens which may be connected to processes, and the user may select for viewing any virtual screen by using a sequence of keystrokes. This interface was developed by Mr Glen Pringle.

Support for a graphical interface is planned. This will require modification of the kernel by the addition of code to control the mode of the graphics card.

At present the text based interface interacts with a shell which supports the execution of programs and basic object management, using a simple interactive command language. Where traditional operating systems use a hierarchical directory structure, this shell employs a set based representation directory structure. Mr Glen Pringle is continuing work on the shell.

[1]High system load or unmounting a disk may result in rent collection operation being delayed

Unix Compatibility Library

To allow the exploitation of the significant quantities of existing C code available to UNIX systems, a partial emulation of the C libraries which provide an interface to UNIX has been constructed.

The input-output and memory allocation functions of the standard C library are based on the functionality found in UNIX operating system. As the Walnut Kernel is based on a different paradigm, code which emulates the features of UNIX using the functions provided by the Walnut Kernel has been written. This work has been only partially completed.

Mr Carlo Kopp's work on a library that emulates the functionality provided by UNIX for standard input-output is nearing completion.

Professor Chris Wallace, Mr Glen Pringle and myself have developed a partial emulation of the standard C memory allocation routines. Work is continuing in this area.

Work is also continuing on the development of a replacement for standard C libraries where existing libraries are unable to operate in the environment provided by the Walnut Kernel.

Shared Libraries

The Walnut Kernel was designed to facilitate the sharing of memory objects. The development of shared libraries under this environment presents a number of challenges. It is necessary to provide mechanisms which allow the relocation of code, the dynamic linking of code and support the storage of variables local to the instance of the shared library being used. In addition it is necessary to consider policies for updating and possibly removing old libraries.

Preliminary work has been performed to demonstrate possible mechanisms for implementing shared libraries. This work exploited the i486's support of position independent code.

A number of possible approaches to the problem of implementing shared libraries have been put forward. These include:

Locating modules of the libraries at a **fixed virtual address**. This requires

programs which use modules from these libraries to load the module and any pages required to store data local to the instance of the module at a fixed location in the process's address space. This mechanism is simple to implement and allows the most efficient addressing modes available on the processor to be used. It has significant disadvantages in that it places restrictions on the organization of program memory, and may prevent modules from multiple libraries being used in the same program due to clashes in address space usage.

Where the processor provides support for position-independent code, a **table of pointers** located at a known place in the process's address space can be used to access data items and subroutines which are not contained within the code module. The table can be indexed with the capability index allowing the shared routines to be arbitrarily placed in the available address space. This mechanism offers flexibility in the structuring of process address space and does not prevent modules from multiple libraries being employed. The use of indirect access to data and code outside the shared module reduces the efficiency of the code.

An alternative mechanism for use on systems supporting position independent-code is to employ a **buddy scheme**. This method places a data object at a fixed distance away from the code object. This object would contain all references and local storage for the instance of the shared module. This scheme offers the majority of the features of the table-based mechanism described above while surrendering only a marginal degree of flexibility in the organization of the process address space. The mechanism also allows faster addressing modes to be used. This is because code can be generated in the buddy page that makes most efficient use of the available modes while allowing the code to be customised for access to data in the current instance of the shared library.

Professor Chris Wallace has students currently investigating the implementation of shared libraries under the Walnut Kernel.

With suitable language support it is possible to dynamically replace a shared library. Languages such as Erlang[2] provide mechanisms which allow the most recent version of a function to be called in preference to the version existing at the time of

[2] A functional language developed at Ericsson and Ellemtel Computer Science Laboratories [AVW93]

starting the program. More traditional languages, such as C, do not provide direct support for the replacement of a module.

To support the dynamic replacement of modules, without language support, it is necessary to check for the existence of a new module each time a function in the module is called. This process is potentially expensive and careful evaluation of the costs and benefits of providing dynamic replacement of modules is required.

12.2 Hardware

Chapter 11 described the design of the proposed hardware and described work currently being undertaken by Dr Ronald Pose of the Department of Computer Science, Monash University and his students. This work covers the implementation of hardware based on the principles outlined in [CP94] and VLSI-based distributed arbitration circuitry.

To optimise performance on the environment presented to the operating system by this hardware requires the operating system to dynamically relocate both processes and data pages. To provide best performance it is necessary to balance the competing goals of increasing parallelism by employing a greater number of processors and decreasing the overhead of interprocess communication by limiting the number of processors. The wide range of memory access speeds provided by this design makes this optimisation complex. It is a significant research area in its own right.

Recent research by Dunning and Ramakrishnan of Bowling Green State University [RD93] has shown that the assignment of tasks to a multiprocessor system of moderate size is an NP-complete problem. It is expected that work on assignment of tasks will rely on heuristics and concentrate on the avoidance of worst case performance.

The original intention was to implement the Walnut Kernel on the proposed hardware. However, to speed development, a i486 based platform was selected. A port of the Walnut Kernel to the proposed hardware is still intended.

Chapter 13

Conclusion

The Secure RISC Architecture project is the successor to the Password-Capability System. The features of the kernel on the earlier system have been retained. The Walnut Kernel, however, incorporates significant innovations and alterations to enhance the applicability of the kernel to a wide range of commonly available hardware and to meet the needs of programmers. The hardware component of the project is scalable from a single processor system to a cluster of multiprocessors. The wide scaling range is gained by abandoning centralised clocks and distributing switching hardware. This chapter identifies the features of the Walnut Kernel and of the proposed hardware and evaluates them in the context of other systems.

The design of the Walnut Kernel incorporates significant changes to the original password-capability model to allow a more portable implementation. It demonstrates that a useful system can be constructed using the password-capability model which is easily transportable between hardware platforms. Furthermore, the Walnut Kernel showed that portability has not been won at the cost of security.

Significant departures from the Password-Capability System include moving from a system based on capability registers (a segment register like mechanism) to using paging for all access control; the introduction of the subprocess mechanism; the ability to **restrict** the rights of a capability after it has been created; and the introduction of the **SRMULTILOAD** mechanism and the ability to create capabilities with non-random passwords.

The shift from capability registers to a page-based implementation has made the

granularity of protection coarser. While fine grained control has been lost, it is not crippling. This change forces programmers to use capabilities which cover larger regions of memory than those used in the Password-Capability System. If space is at a premium, programmers can aggregate small data structures, which have the same security requirements, into a single object. Offsetting the coarsening of the protection domain is the ability to load and access a far greater number of objects at one time. This reduces the number of load and unload operations required, in comparison to the Password-Capability System, and allows the programmer greater flexibility.

Difficulties found while writing application programs have motivated changes to the functions provided by the kernel.

The message passing mechanism was required to handle messages of different types in different ways and to guarantee the delivery of high priority messages. This requirement was met by introducing a subprocess mechanism, and functions which allow the reservation of mail boxes based on target subprocess or message prefix. The subprocess mechanism provides elegant handling of events asynchronous to a process. The mechanism can also be used to implement a form of co-operative multitasking within a process.

The initialisation process requires a capability for the physical memory where the kernel resides. Although the passwords for this capability are randomly allocated, this situation presents a significant opportunity for attacking the system. To improve security it was useful to be able to remove rights, through the **restrict** operation, from this and other capabilities. The ability to remove rights from a capability after it has been created is a fundamental change from the Password-Capability System. Access to this capability would allow a process unfettered access to the system. The **restrict** operation deletes rights from a capability preventing those rights being used by processes that load the capability after the **restrict** operation has been performed. Instances of the capability loaded at the time of the **restrict** operation are unaffected. The operation enhances the security of the system by pruning rights present in the capability tree.

While writing the screen and keyboard device manager - **Glui** - it was found

that device managers needed greater control over processes using capabilities which allow access to raw devices. A new mechanism to control the scheduling of a process was introduced, along with a mechanism to limit the use of a capability to a single process. Capabilities without the **SRMULTILOAD** right can only be loaded by processes with a serial number equivalent to the password 2 of the capability. This mechanism provides a a method of restricting the use of a capability to a specific process. This mechanism is teamed with the **protected freeze** and **thaw** operations. These operations allow a process to be frozen and to prevent another process from unfreezing it.

To make the **SRMULTILOAD** right useful it was necessary to add the ability to specify the passwords of a derived capability. This mechanism increases flexibility, as it allows capabilities with widely known values to be used as mechanisms for accessing services, and it does not reduce the security of the system

Another significant difference from the Password-Capability System is that part of the serial number of a capability under the Walnut Kernel is used to represent the disk block of the header block of the object. This change has been shown to have a minimal cost in terms of the ability to guess a capability. At most one bit of the serial number is lost.

The current fixed queue size round robin scheduling mechanism is inadequate for a production system. However, a user process providing storage for the names of processes to be scheduled and long and medium termed scheduling is planned to overcome the deficiencies of current scheduler.

Two types of windows have been introduced into the Walnut Kernel. This makes the programmer's task more complex. All capabilities can be loaded into small windows but only a small fraction of the process address space is available for small windows. Some capabilities cannot be loaded into a large window. The convenience of the size of large windows is traded for the flexibility of small windows.

Execute only code is not available under the Walnut Kernel as its target population of processors does not generally support execute without read in page table entries. This prevents the use of embedding capabilities in execute only code as a mechanism of protecting capabilities. Furthermore, it prevents the hiding of algo-

rithms and implementations from users of library code. Mechanisms that provide these protections using the available facilities are being investigated.

The Walnut Kernel does not have an adequate backup mechanism. Currently volumes can be archived at the block level and restored after a failure of the media. However, although individual users can backup the contents of an object they cannot backup an object itself. The set of capabilities derived for the object and the capability for the object is lost. The partial backup and restore problem is common to most capability based systems. In addition, there is a need to eliminate tampering with the contents of a backed up object to prevent manipulation of the money field.

Unlike the Password-Capability System, which was designed around custom hardware, the Walnut Kernel was written with portability as a high priority. The design traded the advantageous features of a specific architecture for a more generic set of functions. In the case of the i486 implementation this meant sacrificing the fine grained control offered by segment registers for the more widely available paging mechanism.

The Walnut Kernel incorporates a general mechanism for accessing devices by representing the interfaces to devices as operations performed on blocks of memory mapped into the kernel's address space. This model of interaction between the kernel and hardware exactly describes the actions of a multiprocessor system with specialised IO controllers which write into shared memory blocks. It is also a good fit for uniprocessor versions of the kernel as a timer interrupt or a device interrupt can be used to activate an interrupt service routine which treats the memory block as memory shared with the kernel. This model has the valuable side effect of allowing interrupt service routines to be written as if they are separate from the kernel. This simplifies the task of porting the kernel from system to system as the device drivers are clearly identifiable and independent of the kernel code.

A simple but effective memory management model was adopted for managing the physical memory. Part of the physical memory is reserved for the kernel. The kernel is loaded into this section of memory and remains resident. The kernel area is not paged and is present in all process address spaces. The physical memory used

to hold pages of objects is managed by allocating new pages - on demand - from the list of free pages. Pages are added to the free list by a procedure in the kernel which periodically invalidates and disposes of clean pages, and writes dirty pages to disk. The simple expedient of periodically disposing of page tables and maintaining a last reference time for each page of an object allows the kernel to ensure that a page cannot be accessed and can safely be removed from the physical memory.

The Walnut Kernel allows a far greater number of capabilities to be loaded by a process than the Password-Capability System. Also, the address space of a process is considerably larger in the Walnut Kernel. These facilities are partial compensation for the loss of fine grained access control that was provided by specialised hardware.

The performance measurements indicate that the Walnut Kernel provides similar performance to a conventional operating system, where the operations are comparable. This demonstrates that capability based operating systems in general and the Walnut Kernel in particular could become practical alternatives to conventional operating systems.

The survey clustered operating systems into categories based on the kernel type and the dominant paradigm of the kernel. From a commercial perspective the most successful operating system has a monolithic kernel and has a file based paradigm. Small kernel and micro-kernel based operating systems have emerged as both production and research systems. Capability based operating systems tended to be experimental.

KeyKOS demonstrated that a capability based operating system offered features and facilities demanded by commercial computing users. Of particular importance to the commercial environment were the high level of security, ease of sharing data and the ability to accurately charge for services. Furthermore, a capability based operating system provided these features as a natural extension of the operating system.

The concepts of small kernels and micro-kernels are valuable as they allow software engineering practices to be applied to systems programming. Critical pieces of code are isolated, and well defined interfaces are used for communication between both kernel components and layers of software providing services.

The Walnut Kernel was designed as a small kernel to provide the functionality required in a commercial environment. The password-capability mechanism is more flexible than the alternatives of segregation and tagging. Tagged architectures are demonstrably not cost efficient, while segregated architectures are more complicated and less flexible. The ability to directly manipulate a capability and treat it as ordinary data eliminates the need for the kernel to mediate all operations on capabilities. If the kernel is required to perform all operations on capabilities the set of operations is limited to those explicitly provided for by the operating system designer. The Walnut Kernel allows user code to extend the range of operations available for manipulating a capability. Examples of where this ability might be useful include communicating capabilities through an untrusted third party. The Walnut Kernel would allow the most up-to-date encryption algorithms to be employed. Alternative systems are either unable to provide this facility or the algorithm is fixed in the kernel.

Single address space operating systems such as Opal provide a more direct link between capabilities and addresses than the load/unload capability model employed by the Walnut Kernel. This simplifies the application of the paradigm and reduces complexity for the programmer. Users of a SASOS list a set of capabilities for the protection domain that the program is allowed to use. Users of the Walnut Kernel are required to explicitly load capabilities into their address space. The Walnut Kernel enjoys greater portability than SASOS operating systems as an address space significantly larger than 32 bits is required to adequately support the SASOS paradigm. The Walnut Kernel functions well in a 32 bit address space.

The container mechanism used in Grasshopper is more general than the loading of segments provided by the Walnut Kernel. Containers are a collection of segments of objects. Containers may be nested within other containers. Under Grasshopper, the locus of a process's execution jumps from container to container as it proceeds. Similar functionality could be emulated under the Walnut Kernel by employing a set of routines within each Walnut Kernel process that load and unload segments based on addressing exceptions. However, the implementation would be less elegant and less efficient.

The hardware proposed is novel in two respects. The hardware distributes the interprocessor switches allowing for cost effective system growth, and it eliminates the requirements for central clocking of the system.

Current multiprocessors, such as the CM-5, use centralised interconnects which allow easy growth up to the capacity of the interconnect. Adding a single processor when an interconnect is full, incurs the cost of adding either a new interconnection module or replacing the existing interconnect. The cost of the interconnect dominates the sizing of the system. By distributing the switching hardware, we allow the interconnect to grow as the system expands. This allows the number of modules to be right sized for the problem, as the cost of the interconnection no longer determines the most cost efficient size of the system.

Eliminating the requirement for central clocking, the multiprocessor allows greater flexibility in the physical layout of the system. In addition, it lifts size constraints on a multiprocessor allowing systems to be constructed using multiple cages, and even cages separated by large physical distances. This allows systems to be easily put together for large jobs. The mechanism proposed has the additional benefit of allowing processors to operate at different clock speeds within the same multiprocessor. This has significant advantages in that allows investment in earlier model processors to be retained without losing the full benefit of incorporation of faster new processors into the system.

The Secure RISC Architecture project incorporates both hardware and software elements to address the problem of building systems which range in size from single processor workstations through to large multi-cabinet multiprocessors. The Walnut Kernel operating system and the hardware were designed to be mutually supportive to provide performance, security and reliability.

Appendix A

User Level Programmer's Guide

This appendix is drawn from the technical report 95/222 entitled *The Walnut Kernel: User Level Programmer's Guide*. This technical report is available from the Department of Computer Science, Monash University:

```
Title:    The Walnut Kernel: User Level Programmer's Guide
Author:   Maurice Castro
No.:      95/222
Revised:  November 1995
```

I present it in support of my thesis.

A.1 Overview

The Walnut Kernel is a capability-based operating system under development in the Department of Computer Science at Monash University. This operating system draws on the concepts of and experience gained from the Password-Capability System[1].

The Walnut Kernel employs 128-bit names - Password-Capabilities - for views onto persistent objects. The random allocation of names within a sparse name space provides a known level of statistical security for views and the contents of objects. Associated with each name is a set of rights which entitle the holder of the capability to access a section of the named object in a specified way.

The Walnut Kernel was designed as a portable operating system although it currently runs only on 80486 based PCs. Programs are compiled on a FreeBSD 1.1 system and transferred onto the target machine on floppy disks. Work is continuing on the development of the kernel as well as the development of interfaces, shells, and utilities for the system.

This document contains a programmer's manual for the Walnut Kernel. The document is subject to revision as the kernel alters and currently describes only the lowest level of the kernel interface.

Acknowledgments

The author would like to acknowledge the following contributions:

- Prof C.S. Wallace - co-author of the kernel
- Mr Glen Pringle - author of many of systems utility programs
- Mr Carlo Kopp - author of the UNIX compatibility libraries

The 'Secure RISC Architecture' project is supported by a grant from the Australian Research Council (A49030623). Maurice Castro is a recipient of an Australian Postgraduate Research Award.

[1] M. Anderson, R D. Pose, and C S. Wallace. A Password-Capability system. *The Computer Journal*, 29(1):1–8, 1 1986.

A.2 Objects

All entities controlled by the Walnut Kernel are objects. A Walnut Kernel object is analogous to a segment in segmented computer architecture. It comprises an ordered array of bytes. An individual byte is identified by its 'offset', a number indexing the array. The first byte has offset zero. An object is defined by the following characteristics:

Maximum Offset The largest addressed offset in the object. (Note: this value is set on creation and automatically increases, as long as there are pages available in the allocated space of the object)

Limit The largest addressable offset allowed in the object.

Maximum Size The maximum number of bytes guaranteed to be available to an object. The number of bytes includes storage for the objects capabilities and dope vectors. (In practice this value represents the maximum number of pages and header pages guaranteed to be available to an object. The number of pages is calculated by dividing the maximum size by the page size and rounding upwards.)

Maximum Capabilities The maximum number of capabilities that can represent this object. (Note: this value automatically increases, as long as there is space available to hold the new derived capabilities)

Money The amount of money the object has available. Sufficient money must be present in an object to pay for its resource consumption.

Each object has at least one capability that allows access to the object - the object's **Master Capability**. Deletion of the object's master capability results in the deletion of the object. Other capabilities for views of the object are derived from the master capability or its descendants.

A.3 Capabilities

The Walnut Kernel employs password capabilities to identify access rights to objects. Each capability (see figure A.1) consists of a 128 bit identifier composed of four 32 bit values: a volume number, a serial number, password 1 and password 2. Associated with each capability is a view which determines the region of an object a capability applies to, a set of user rights, and a set of system rights which control how that capability is to be used.

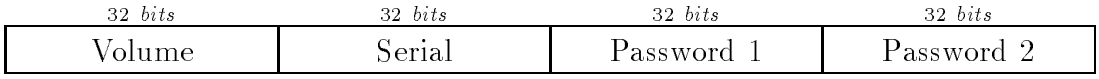


Figure A.1: A Password Capability

A.3.1 View

A view is the attribute of a capability that defines the region of the object that can be addressed by the possessor of the capability. Views are contiguous regions and are defined by an offset from the base of the object and an extent. The view entitles the user to address part of an object, it does not guarantee that pages are contained in that region nor that the pages are readable by user processes.

A.3.2 User Rights

User rights consist of a set of 32 bits which are managed by the kernel. The kernel attaches no meaning to the user rights bits. They are intended to be used by user processes to implement access to services in a way that is analogous to the control system rights bits have over access to kernel services.

A.3.3 System Rights

The system rights associated with a capability are encoded in a 32-bit word, basically as the OR of bits representing particular rights (the SRSEND field is an exception). For the numeric value of the system rights symbols see figure A.4.

SRDERIVE - Allow capabilities to be derived from this capability.

SRSUICIDE - Allow this capability to destroy itself and its children.

SRDEPOSIT - Allow the holder of this capability to deposit money into the object.

SRWITHDRAW - Allow the holder of this capability to withdraw money from the object.

SRREAD - Allow the holder of this capability to read from the view.

SRWRITE - Allow the holder of this capability to write to the view.

SREXECUTE - Not used.

SRUSER - Allow user processes to use the view.

SRPEEK - Allow the holder of this capability to perform a peek system call on the process represented by this capability (see A.11.2).

SRMULTILOAD - Allow this capability to be loaded by any process. If this right is absent then only processes with a serial number equivalent to the capability's password 2 may load this capability.

SRSEND - an 8-bit field which, if non-zero, specifies the subprocess to which messages may be sent by using this capability. This field has two special values: `0xff` - allow messages to be sent to any subprocess of the process, and `0xfe` - disallow messages to subprocess zero but allow messages to be sent to any other subprocess of the process.

A.3.4 Deriving Capabilities

Derived capabilities have equal or lesser rights than than their parent capability, at the time of derivation. *Suicide right* is an exception as this right may be added to the children of capabilities which do not hold this right.

The rights of a parent capability may be reduced through the use of the *restrict* system call after a child capability has been derived. The child capability is unaffected by the restriction of the parent capabilities rights.

A.4 Process Structure

A process in the Walnut Kernel is essentially an object which contains state information relating to the execution of the process. The minimal information found within a process object is:

- Sub-process table
- Message slots
- Table of Loaded Capabilities
- Process cash
- Lock words
- Parameter page
- Address map

Of these only the parameter page and the address map are directly accessible to the user process. The address map is read-only. The parameter block and the remainder of process object are both readable and writable by the user process.

The process structure is detailed in diagram figure A.2.

A.4.1 Process Address Space

The address space of a process operating under the Walnut Kernel is composed of three regions:

Kernel Area is located at the bottom of the address space and is not addressable by user processes.

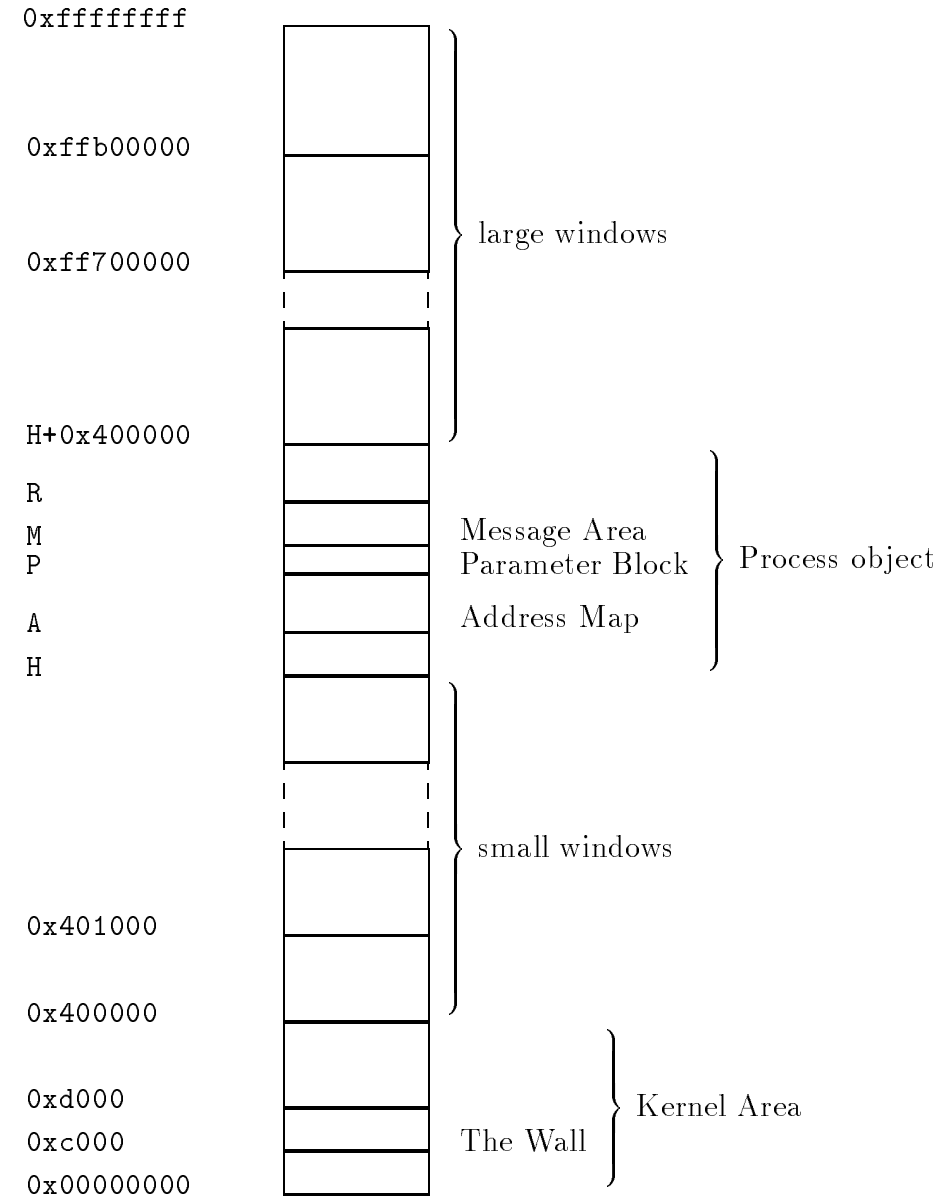
Small Window Area is located above the Kernel Area and has a page sized granularity. Single pages or multiple pages of objects may be mapped into this region of the address space by a user process. These mapped regions always begin and end on a page boundary.

Large Window Area is located above the Small Window Area. It has a coarser granularity than the Small Window Area. The first large window contains the Process Object. All other large windows are allocated by the process.

On a system with a 4 kilobyte page size, large windows have a granularity of 4 megabytes and small windows have a granularity of 4 kilobytes.

Two distinct paradigms are used to describe how the address is populated with objects.

The Password-Capability system used the term 'Window Registers' to describe a set of segment registers. The Password-Capability system used the upper bits of



	Description	Constant Name	Value
R	remainder of process object	PARAMADDRESS + 0x1000	0x1011000
M	message area	EXTRAADDRESS	0x101004c
P	parameter page	PARAMADDRESS	0x1010000
A	address map		0x100f000
H	process header	PROCHDADDRESS	0x1000000

Figure A.2: **Process Address Space**: This diagram describes the major features of the address space seen by a process operating on a system with 4 kilobyte pages. The message area and the parameter block are collectively known as the parameter page.

the virtual address to indicate which register was in use. We retain this terminology in the Walnut Kernel. The analogy between the two systems is imperfect as: the Walnut Kernel supports two classes of window registers; and although the number of window registers is fixed on the Walnut Kernel the location of the registers in the virtual address space is not fixed, under the Password-Capability System both of these parameters were constant.

The second paradigm describes the operation of the system. On a system with 4 kilobyte pages, it views the address space as two address ranges. The address range from `0x400000` to `0xffffff` can have objects loaded on 4 kilobyte boundaries. The address range `0x1400000` to `0xffffffff` has objects loaded on 4 megabyte boundaries.

A.4.2 Parameter Page

This page is composed of two parts: the parameter block and the message area. The parameter block is a structure defined in the file **param.h**. This block is used to pass parameters to the kernel when a system call is made. The second area is used to pass additional information to the kernel and to receive information from the kernel. The information passed via the message area varies with the type of call.

Parameter Block

The declaration of the parameter block structure is found in figure A.3.

The fields are named after the function they are used for in the majority of calls. The values contained in the fields are:

error The error field contains an integer error value on returning from a system call. If the value is zero then the system call completed successfully. If the value is greater than zero the system call could not be completed successfully. If the return value is negative[2] or greater than 20000000_{10} an internal kernel error has occurred: contact the system's maintainer urgently and report the value. The file *include/kerror.h* contains a translation table which allows error values to be converted to ascii strings.

vol serial pass1 pass2 The capability field composed of vol, serial, pass1 and pass2, contains either a capability being passed to a system call or a capability being passed back by a system call.

srights The system rights field contains one of

- a bitmap indicating the system rights provided by a capability (Figure A.4 defines the symbolic and numeric forms of the system rights bits)
- a mask which restricts the system rights provided to a derived capability

[2]Negative error values are used internally by the kernel to indicate partial completion of a system call which cannot be completed because of a transient problem.

```
/* ------------------------------------------------------------ */
/*                      Parameter Structure                     */
/* ------------------------------------------------------------ */
typedef struct Paramst {
    Sw error;
    Uw vol;          /* volume ID */
    Uw serial;       /* serial in volume */
    Uw pass1;        /* password 1 */
    Uw pass2;        /* password 2 */
    Uw srights;      /* System rights */
    Uw urights;      /* User rights */
    Sw base;         /* Offset of cap from front of object */
    Sw limit;        /* Max addressing offset from base */
                     /* Zero means "to end of object" */
    Sw money;        /* money word */
    Uw type;         /* Type of object */
    Sw maxoff;       /* Max addressing offset in whole object */
    Sw maxsz;        /* Max size of defined content */
    Sw maxcap;       /* Max capabilities now allowed */
    Sw offset;       /* An offset in a capability window */
    Sw subpn;        /* A subprocess number */
    Sw cindex;       /* Index of a capl in a process TLC */
    Uw clocktime;
    Uw reserve;      /* Non-zero shows reserved by sub-process */
    } Param;
```

Figure A.3: **Parameter Block Declaration**

- a set of limits used in the creation of a process
- an encoded process state when inquiring about a process state

urights The user rights field contains either

- a bitmap indicating the user rights provided by a capability
- a mask which restricts the user rights provided to a derived capability

base The base field contains an offset from the beginning of a view or - on process creation - the time, in seconds, at which the new process is scheduled to wake up.

limit The limit field contains one of

- the length of a message
- the maximum addressable offset of an object
- the maximum size of a view

money The money field contains one of

- the amount of money in an object
- the amount of money to be deposited or withdrawn
- the amount of money to be sent with a message
- the amount of money received from a message

type The type field contains the object type. The top bit of this field is set if the object is a process. The following type values are reserved by the kernel:

`00000000`	
`00000003`	Prototype process
`0000ffff`	Physical memory object
`80000000`	
`80000002`	Drive process
`80000003`	Prototype process

maxoff The maximum offset field contains the current maximum offset of an object.

maxsz The maximum size field contains the current maximum size of an object.

maxcap The maximum capability field contains the current maximum number of capabilities for a process

offset The offset field contains an offset into a process's address space.

subpn The subprocess number field contains the destination subprocess number for a message.

cindex The capability index field contains the index into the table of loaded capabilities that a capability occupies.

clocktime The clocktime field, after returning from a system call, contains the current time in seconds. The clocktime field is set to the wakeup time for a process when a wait system call is made.

reserve The reserve field provides both a locking function which prevents other subprocesses accessing the parameter block and indicates the type of kernel call being made. The currently available kernel call constants are listed in figure A.5.

```
#define SRDERIVE    0x40000000  /*  System rights bits   */
#define SRSUICIDE   0x20000000
#define SRDEPOSIT   0x10000000
#define SRWITHDRAW  0x08000000
#define SRREAD      0x04000000
#define SRWRITE     0x02000000
#define SREXECUTE   0x01000000
#define SRUSER      0x00800000
#define SRPEEK      0x00400000
#define SRMULTILOAD 0x00200000
#define SRSEND      0x000000FF  /* Bits relating to send rights */
```

Figure A.4: **System Rights Constants**

Message Block

The message area's contents are interpreted differently for each class of call. There are currently three classes of information stored in the message block:

- Messages - Messages to be sent by the send message system call and messages recovered by the receive message system call are stored at the front of the message block.
- System states - The save register and load register system calls store the register set and other state information for a subprocess at the front of the message block.
- Read/Write Data - Bytes to be transferred by the external read or external write system calls are stored at the front of the message block.
- Initialization information - Initial values used to set stack and program counters, the name of an heir and a list of capabilities to be pre-loaded into a process are stored at the front of the message block.

```
/* ------------------------------------------------------------ */
/*                          Action Codes                        */
/* ------------------------------------------------------------ */

#define K_MAKEOBJ 1
#define K_MAKECAP 2
#define K_DEL 3
#define K_DELDER 4
#define K_RESIZE 5
#define K_SHRINK 6
#define K_WAIT 7
#define K_LOADCAP 8
#define K_UNLOADCAP 9
#define K_CAPID 10
#define K_MAKEPROC 11
#define K_SEND 12
#define K_RECV 13
#define K_EXTSEND 14
#define K_EXTREAD 15
#define K_EXTWRITE 16
#define K_BANK 17
#define K_RESTRICT 18
#define K_CAPSTAT 19
#define K_RENAME 20
#define K_MAKESUBP 21
#define K_DELSUBP 22
#define K_LOADREG 23
#define K_SAVEREG 24
#define K_SETTRAP 25
#define K_RECV_CLOSE 26
#define K_ACCEPT_MAIL 27
#define K_CLOSE_BOX 28
#define K_COPYOBJ 29
#define K_PEEK_PROC 30
#define K_SET_HEIR 31
```

Figure A.5: **Defined Kernel Call Constants**

A.4.3 The Wall

Every process has a read-only page mapped into its address space known as **the Wall**. This page contains public information, including the current time, and the capabilities of public utilities. A **wall manager** places information in the wall. The wall currently contains:

`0xc000` Scheduler Start Variable†
`0xc004` Physical Object: Volume†
`0xc008` Physical Object: Serial†
`0xc00c` Physical Object: Password 1†
`0xc010` Physical Object: Password 2†
`0xc000` GLui: Magic Number
`0xc004` GLui: Volume
`0xc008` GLui: Serial
`0xc00c` GLui: Password 1
`0xc010` GLui: Password 2
`0xc014` Name Server Set: Magic Number
`0xc018` Name Server Set: Volume
`0xc01c` Name Server Set: Serial
`0xc020` Name Server Set: Password 1
`0xc024` Name Server Set: Password 2
`0xc028` Name Server Set: Offset
`0xcfe0` Time in Seconds
`0xcfe4` Time in Microseconds

†: These locations are used by the initialization process. After initialization the capability of the physical object is overwritten by the initialization process and the value of the scheduler start variable is no longer significant.

A.5 Process Structure Conventions

This section covers the conventional layout of a process (Section A.4 outlined the mandatory elements of a process structure).

A.5.1 The Process Object

The following elements of the process object are visable to the user process and are provided by the kernel:

- The Process Address Map
- The Parameter Page
- The Message Area

They form part of the mandatory component of the process structuring convention used by Walnut Kernel processes.

By convention the following items are located within the process object

Startup Code Area (optional) This area may contain a small amount of code used in starting a process.

File Descriptor Table (mandatory) This area contains the file descriptors for use by the process. Note: The first 3 elements of the File Descriptor Table are mandatory to allow for standard output, standard input and standard error.

Private Data Pointer Table (mandatory) This area contains pointers to private data. The table is indexed by the capability index of the executing code and is used to locate data used by the executing code.

Default Heap (optional) The default location for the creation of the heap.

Default Stack (optional) The default location for the creation of the stack.

The structure of the process object is outlined in figure A.6.

A.5.2 The Process

Conventional processes will be constructed according to the following rules:

- The code object will be loaded at address `0x1400000`
- The data object will be loaded at address `0x5400000`
- Initialized data will be placed at the front of the data object

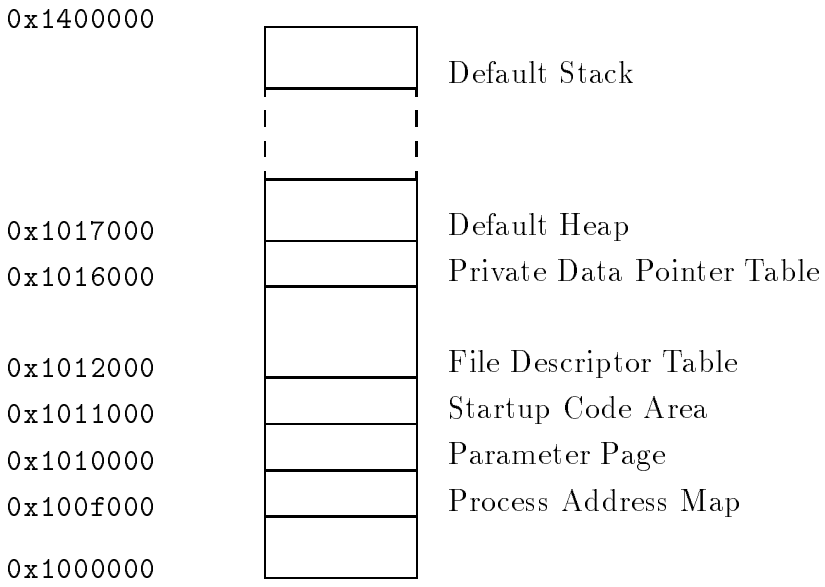


Figure A.6: **Process Object**: This diagram describes the major features of the process object

This design allows multiple instances of a process to be created by sharing the code objects and using copies of the data objects. In addition by placing the initialized data at the front of the data object it is possible to ensure that the original data object is compact and hence easy to copy. The copy of the data object will expand as required when uninitialized data is accessed.

This arrangement of code and data allows up to 64 Mbytes of code to be supported. With the introduction of shared code libraries larger programs can be supported.

A.6 Process Creation

This section describes the process of creating a process and the initial state of a new process.

A.6.1 Making Processes

A process is created using the **Make Process** call covered in section A.11.2. This section will provide a general introduction to the creation of a process.

Creating a process involves:

- Creating a new process object
- Creating an address space
- Loading the new process object into the address space
- Creating subprocess 0 and subprocess 1.
- Loading pre-loaded capabilities into the address space
- Setting initial program counter and stack pointer values for subprocess 1.
- Setting the wake up time of subprocess 1.
- Loading the new process object into address space of the creating process

This process appears as an atomic operation to the process issuing the **Make Process** system call. If the system call was successful the master capability for the new process object will be returned and the new process will be loaded at the address given in **offset** in the parameter block.

At the completion of the **Make Process** system call, the new process object is loaded into the address space of the creating process. If the new process object is larger than 4 megabytes in size, only the first 4 megabytes of the new process object is visable. Thus the process which issued the **Make Process** system call and the new process have a region of shared memory.

A.6.2 Initial Process State

Immediately after a process has been created:

- The parameter block of the new process will contain:

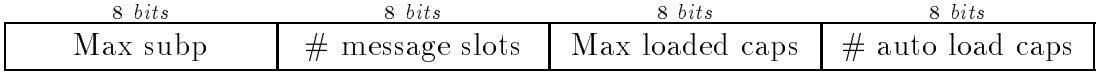
vol	Volume of master capability for process
serial	Serial of master capability for process
pass1	Password 1 of master capability for process
pass2	Password 2 of master capability for process
srights	Encoded process creation parameters
urights	User rights of process's master capability
limit	Maximum size of process object (hard limit)
money	Amount of money in process object / process cash
type	Type of process
maxoff	Maximum offset of view on process object
maxsz	Maximum size of process object
maxcap	Maximum number of capabilities

- The message area will consist of a table of pre-loaded capabilities with the format:

vol	Volume
serial	Serial
pass1	Password 1
pass2	Password 2
base	Start of the loaded window relative to the capability
limit	Size of the loaded window. Zero indicates capability limit
offset	Location of window in the new process's address space
cindex	Index in table of loaded capabilities. Zero for automatic allocation

- The process object will be loaded at the location PROCHDADDRESS
- The address space will contain all pre-loaded capabilities
- Only subprocess 0 and subprocess 1 will exist
- Subprocess 1 will begin executing

The process creation parameters are encoded in the system rights field:



msb lsb

- Max subp - The maximum number of subprocesses for the new process including subprocess 0.
- # message slots - The number of message slots for the new process. As a message slot is reserved for subprocess 0 the number of message slots must be 1 or greater.

- Max loaded caps - The maximum number of loaded capabilities for the new process. This number includes the capability for the process.

- # auto load caps - The number of capabilities to be automatically loaded into the new process's address space including the capability for the new process.

When a process is created two equal sums of money are deposited into the new process. The sums are deposited into the process cash and the process object respectively. The size of one of the deposited sums is reported in the money field.

It is normal practice for the first action of a process to be the duplication of the information passed in at process creation. It is particularly important to store the capability for the process as it is not possible to locate the master capability for the process subsequently.

The creation of subprocesses other than subprocess 0 and subprocess 1 is handled by the application using the Make Subprocess call.

A.7 Subprocess Zero

The Walnut Kernel implements two direct methods of communication with the kernel: system calls and messages to subprocess zero of a process. The system call mechanism (described in the section A.11) allows a process to alter its own state, operate on capabilities and send messages. The subprocess zero mechanism allows a process to control another process's state.

Subprocess zero functions are accessed by sending messages to a process's subprocess zero. The message contains a function identifier and arguments. On receipt of a message to subprocess zero the kernel interprets the instruction provided and performs the required action. Subprocess zero operations and messages are the highest priority function of a process.

The currently implemented subprocess zero functions are:

Freeze Prevent process from being scheduled.

Thaw Allow process to be scheduled.

Wakeup Set the wakeup time of the specified subprocess to zero

Cooee Request the process to send a status message using a specified capability.

Protected Freeze Prevent process from being scheduled until all protected freezes on the process have been thawed.

Protected Thaw Allow a process to be scheduled when all other protected freezes have been thawed.

Figure A.7 lists the identifiers and arguments of the messages.

Function	Function ID	Arg 1	Arg 2	Arg 3	Arg 4
Freeze	33330001				
Thaw	33330002				
Wakeup	33330003	subp #			
Cooee	33330004	vol	ser	pass 1	pass 2
Prot Freeze	33330007	magic			
Prot Thaw	33330008	magic			

Figure A.7: Subprocess Zero Functions and Arguments

A.7.1 Freeze

On receipt of a freeze message subprocess zero sets the process state to frozen and causes the process to be removed from the scheduler queue.

A.7.2 Thaw

When a process receives a message it is placed into the scheduler queue. If the process is frozen the process is typically removed from the queue after the subprocess zero messages are parsed. On receipt of a thaw message, subprocess zero sets the process state to normal and process execution resumes.

A.7.3 Wakeup

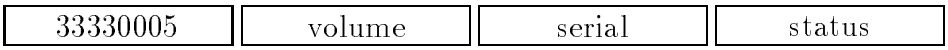
The wakeup message sets the wakeup time of the nominated subprocess to the current time. This allows a process to start a process that has suspended activity and has closed mail boxes as the mail box allocated to subprocess zero cannot be closed.

One application of this function is to allow the initialization of data structures within a process object. The process is created with a wakeup time of never preventing the scheduling of the process. The creating process initializes the required data structures before waking the created process up. At that stage the created process may elect to open its mail boxes as processes are created with all but subprocess zero's mail boxes closed.

A.7.4 Cooee

On receiving a cooee message subprocess zero attempts to send a message using the capability found in the cooee message. If the capability in the cooee message allows transmission to any subprocess of a process then the message will be sent to subprocess one of the nominated process, otherwise, the message will be sent to the subprocess represented by the capability.

The reply message is of the form:



The message consists of a set of words which represent the Cooee reply identifier, the volume and serial number of the current process and a process status. The process status is given in figure A.8.

State	State ID	Value
Normal State	PROCSTATENORMAL	1
In Kernel Call	PROCSTATEKERNEL	2
In Read Fault	PROCSTATERFAULT	3
In Write Fault	PROCSTATEWFAULT	4
Process Frozen	PROCSTATEFROZEN	5
Process in Probate	PROCSTATEPROBATE	6
Process Dead	PROCSTATEDEAD	7

Figure A.8: Process Status

A.7.5 Protected Freeze

On receipt of a protected freeze message subprocess zero sets the process state to frozen, XORs the magic word with a key held in the process state, increments a count held in the process state and causes the process to be removed from the scheduler queue. This prevents other parties from thawing the process unless they know the set of magic words used in the protected freeze operations applied to the process.

A.7.6 Protected Thaw

On receipt of a protected thaw message subprocess zero XORs the magic word with a key held in the process state and decrements a count held in the process state. If both the count and key held in the process state are zero then the process is thawed. If the count is zero and the key is non-zero then the process is terminated.

A.8 Subprocesses

Subprocesses are implemented in the Walnut Kernel as threads of execution which share a single address space. This section describes subprocesses and their scheduling.

A.8.1 Anatomy of a Subprocess

When a process is created a fixed number of subprocess slots are allocated in the process structure. These slots form the *subprocess table* which is used to store the subprocess states.

When a subprocess is created the creator specifies a priority which is used to determine which subprocess should be scheduled, the starting address of the subprocess and the the address of the subprocess's stack pointer. It is the responsibility of the programmer to ensure that the stacks of subprocesses do not overlap.

Subprocesses share the address space of the process and hence have no protection from the actions of other subprocesses of the process.

A.8.2 Operations on Subprocesses

Subprocesses can be made through the use of the **K_MAKESUBP** system call and they are destroyed by **K_DELSUBP**. Messages are sent to subprocesses using **K_SEND** and **K_EXTSEND**. There are three types of capabilities which can be used to send messages: capabilities which can send messages to any subprocess of a process, capabilities which can send a message to any subprocess of a process other than subprocess zero, and capabilities which can only send messages to a particular subprocess. The type of capability determines if the subprocess parameter of the send operation is used.

A.8.3 Scheduling

Subprocesses have the semantics of processes on a time sharing system. That is, when a subprocess of a process is executing no other subprocess of that process can be executing. On the Walnut Kernel processes are used to support concurrent execution.

The algorithm for determining which subprocess to run at the beginning of a time slice for a process is as follows:

1. If a subprocess was executing and there is a non-zero value in the *reserve* field of the parameter block resume execution of that subprocess.

2. Execute the subprocess with the highest priority which is not waiting.

3. For subprocesses of equal priority select the first subprocess encountered in the subprocess table.

Before performing the algorithm to determine which subprocess to schedule the mail boxes are scanned. If a new message has arrived for a subprocess the subprocess is made runnable (not waiting).

Subprocesses can ensure that other subprocesses of the current process are excluded from executing by setting the *reserve* field to a non-zero value. It is essential that any subprocess attempting to make a system call sets the *reserve* field to the appropriate value for the system call before accessing other elements of the parameter block. It is also necessary to test or copy all required values from the parameter block before zeroing the *reserve* field after returning from a system call.

A.9 Messages and Mailboxes

This section describes the processes of sending and receiving messages.

A.9.1 Sending Messages

Messages are sent using either the **K_SEND** or **K_EXTSEND** system calls. A message consists of the contents of the *message area*. The length of the message is variable (currently up to 16 words may be sent) and it is specified by setting the *limit* field to the number of bytes to be transferred.

Messages are sent to processes represented by a capability. The capability may be derived to allow messages to be sent to only one subprocess or to allow messages to be sent to all subprocesses of the process. If the latter type of capability is used then the *subpn* field contains the destination subprocess number.

A message will only be sent if there is an empty mailbox available to receive the message at the destination process. An error is returned if there are no suitable mailboxes at the destination process.

A.9.2 Receiving Messages

Messages are retrieved and mailboxes are cleared by issuing a **K_RECV** system call. A match string can be specified for the *receive* system call allowing the user program to control the order in which messages are retrieved from mailboxes.

When there is a message in a mailbox waiting to be received, the wakeup time of the subprocess is set to the current time. This nullifies the effect of any **K_WAIT** system calls.

A.9.3 Mailboxes

Mailboxes have 3 independent parameters which determine whether or not they will accept a message: state, prefix, and subprocess.

The state of the mailbox:

Open - The mailbox is prepared to accept a message that meets the other criteria

Closed - The mailbox will not accept messages

A message prefix consists of a string of characters:

Non-zero length - Only messages starting with the prefix string are accepted. The length of the prefix string is specified in bytes.

Zero Length - Accept any message meeting the other criteria

Mailboxes may accept messages for specified subprocesses:

Subprocess 0 - 250 - Only messages intended for the specified subprocess are accepted

Subprocess 255 - Accept any message meeting the other criteria

If a message matches the mailbox's criteria and the mailbox is empty then the message is placed in the mailbox. The criteria are used to ensure that mailboxes are available for particular types of messages. The first available mailbox that accepts the message is used.

The **K_RECV_CLOSE**, **K_ACCEPT_MAIL** and **K_CLOSE_BOX** system calls are used to manipulate the parameters of the mailbox.

Both the **K_CLOSE_BOX** and the **K_RECV_CLOSE** system calls close mailboxes. The **K_RECV_CLOSE** receives a message from a mailbox and then closes the mailbox from which the message was extracted. The **K_ACCEPT_MAIL** system call opens a mailbox and specifies the parameters which determine the messages the mailbox will accept.

A.10 Exceptions

This section describes the handling of exceptions by processes under the Walnut Kernel. The default behavior of the Walnut Kernel is to terminate any process which encounters an exception. This behavior can be modified by using *trap handling* subprocesses.

A.10.1 Types of Exception

The Walnut Kernel detects the following exceptions:

FPFAULT - All exceptions relating to errors in arithmetic. This may include floating point exceptions, integer arithmetic exceptions, dividing by zero, overflow and underflow. The types of errors detected by this exception are processor dependent.

OPFAULT - This exception is raised when an invalid instruction is parsed by the processor.

ADDRSFAULT - This exception is used to catch all errors relating to addresses. It is raised under the following conditions:

- An unmapped region of the address space has been accessed
- A write has been attempted on a read-only area of memory
- A read or write has been attempted on an privileged area of memory
- An object could not be automatically expanded to accommodate the attempted access due to the lack of unreserved space on the volume

DBFAULT - This exception is raised whenever a debug exception is raised by the processor. This exception is processor dependent.

ALIGNFAULT - This exception is raised on unaligned accesses. This exception is processor dependent.

A.10.2 Trap Handling Subprocesses

A trap handler is a normal subprocess which has been nominated to receive trap messages for a given subprocess. The **K_SETTRAP** system call is used to inform the kernel where trap messages should be sent. The *set trap* system call takes two arguments: the subprocess for which traps are to be handled and the subprocess which will handle the trap.

A subprocess cannot handle its own traps. If a subprocess traps and the trap message is to be sent to the same subprocess then the process will be terminated.

When an exception occurs in a subprocess, which has a nominated trap handler, the subprocess with the fault is marked **DEAD**, its wake up time is set to **NEVER**

and a message is sent to the trap handler. The format of the message is discussed in section A.10.3.

The trap handler can examine and alter the state of the dead subprocesses register sets through the use of the **K_LOADREG** and **K_SAVEREG** system calls. The subprocess can be restored to operation through the use of the **K_MAKESUBP** system call.

A.10.3 The Trap Message

A five word message is sent (see figure A.9) to the trap handling subprocess. The words of the message are:

1. Message Type - this word indicates that the message is the result of an exception. The *failure message identifier* is `0x3333ffff`.
2. Subprocess Number - the subprocess number of the subprocess in which the exception occured.
3. Fault Identifier - a code which identifies the type of exception which occured (see table A.1).
4. Processor Error Code - a processor dependent error code for non-floating point operations.
5. Floating Point Error Code - a processor dependent error code for floating point operations.

The error codes are processor dependent and are only returned where relevant to the cause of the exception.

`0x3333ffff`	Subprocess Number	Fault Identifier	Processor Err Code	FP Err Code

Figure A.9: Structure of the Failure Message

Mnemonic	Description	Value
FPFAULT	Floating Point Fault	101
OPFAULT	Opcode Fault	102
ADDRSFAULT	Address Fault	103
DBFAULT	Debug Fault	104
ALIGNFAULT	Alignment Fault	105

Table A.1: Error Identifier Values

A.11 System calls

All system calls implemented within the Walnut Kernel use the parameter block to contain all the parameters of the call. There is only one parameter block per process. To prevent subprocesses from altering the parameter block while another subprocess is setting up or receiving the results of a system call it is essential that the reserve field be set to a non-zero value while a subprocess manipulates the parameter block. Setting the reserve field to a value prevents any other subprocess of a process being run until the reserve field is cleared.

A.11.1 Procedure

How to make a system call:

- Put the call number in the parameter block's reserve field
- Fill in necessary parameters
- Call **system_call()**

After a successful system call has been completed:

- Copy any desired information out of the parameter block
- Set the reserve field to zero

After an unsuccessful system call ($error > 0$)

- Copy the error code and any other desired information out of the parameter block
- Set the reserve field to zero

A.11.2 Available System Calls

This section describes the currently available system calls on the Walnut Kernel and the parameters required for those calls.

Make Object

Name	Symbol	Value
Make Object	**K_MAKEOBJ**	**1**

Input Parameters:

vol	-Volume on which to create object
srights	-System rights
urights	-User rights
limit	-Highest byte offset of object (hard limit)
money	-Initial money
type	-Object type
maxoff	-Highest byte offset of object (soft limit)
maxsz	-Maximum size of object
maxcap	-Maximum number of capabilities including master

Output Parameters:

vol	-Master capability (volume)
serial	-Master capability (serial)
pass1	-Master capability (password 1)
pass2	-Master capability (password 2)
srights	-Master capability (system rights)
urights	-Master capability (user rights)
limit	-Highest byte offset of object (hard limit)
money	-Initial money
type	-Object type
maxoff	-Highest byte offset of object (soft limit)
maxsz	-Maximum size of object
maxcap	-Maximum number of capabilities including master

Description:

This call creates an object of the size specified on the volume specified. The object will have the rights dictated by the **srights** & **urights** field.

Before using the limit value, it is transformed:

$$limit = \begin{cases} BIGLIMIT & \text{if } limit = 0 \\ \text{limit} & \text{otherwise} \end{cases}$$

To create a new object the following preconditions must be met $limit\&0x3ff = 0$, $maxoff \leq limit$, $limit \leq BIGLIMIT$, and $maxsz \leq BIGLIMIT$.

Derive Capability

Name	Symbol	Value
Derive Capability	**K_MAKECAP**	**2**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
srights	-System rights mask
urights	-User rights mask
base	-Offset from the beginning of existing view
limit	-Size of derived view
money	-Drawing limit of capability
subpn	-New password 1 (if *subpn* >= 1024)
cindex	-New password 2 (if *subpn* >= 1024)

Output Parameters:

vol	-Volume
serial	-Serial
pass1	-Derived capabilities password 1
pass2	-Derived capabilities password 2
srights	-Derived capabilities system rights
urights	-Derived capabilities user rights
base	-Cleared by call
limit	-Maximum size of derived view
money	-Drawing limit of capability
type	-Drawing limit of derived capability

Description:

This capability derives a capability from a given capability. The new capability may have weaker rights and/or a smaller view of an object. Note that the suicide right may be added to a derived capability.

Attempts to derive capabilities from a capability without the SRMUTLILOAD right always have the same pass2 as the original capability.

If **limit** is set to 0 then the view of the derived capability will extend from the **base** to the end of the view provided by the original capability.

The following pre-conditions must be met $view.limit \geq base$ and $limit \geq 0$.

Delete Capability

Name	Symbol	Value
Delete Capability	**K_DEL**	**3**

Input Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2

Output Parameters:

Description:

Deletes the capability specified (if the capability has suicide right) and all of its derivatives (if the capability has derive right).

Delete Derived Capabilities

Name	Symbol	Value
Delete Derived Capabilities	**K_DELDER**	**4**

Input Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2

Output Parameters:

Description:

Deletes all of the derivatives of the specified capability (if the capability has derive right).

Resize Object

Name	Symbol	Value
Resize Object	**K_RESIZE**	**5**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
limit	-New limit
maxoff	-New maximum offset
maxsz	-New maximum size
maxcap	-New maximum number of capabilities

Output Parameters:

Description:

Resizes an object to the values given in **limit**, **maxoff** and **maxsz**. If **maxcap** is greater than the current number of permitted capabilities then the number of capabilities is increased, otherwise, **maxcap** is ignored.

Preconditions: **to be specified**

Shrink Object

Name	Symbol	Value
Shrink Object	**K_SHRINK**	**6**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2

Output Parameters:

Description:

Shrinks the object to a size just sufficient to contain its current contents and sets the limits to make this the maximum size of the object. The object's **limit**, **maximum offset**, **maximum size** and **maximum number of capabilities** are altered.

Preconditions: **to be specified**

Wait

Name	Symbol	Value
Wait	**K_WAIT**	**7**

Input Parameters:

clocktime -Wakeup time

Output Parameters:

Description:

Provided there are no outstanding messages this call puts the subprocess to sleep until either a message arrives or the wakeup time has been reached. The wakeup times of **0** and **-1** have special meanings:

0 Surrender the remainder of time slice

-1 Set no wakeup time. Awake only when sent a message

Wakeup times are in seconds and are absolute. Relative wakeup times can be created by adding a value to the time found in **clocktime**.

Load Capability

Name	Symbol	Value
Load Capability	**K_LOADCAP**	**8**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
base	-Offset from start of view
limit	-Size of window to be loaded
offset	-Logical address of load location
cindex	-Capability index

Output Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
srights	-System rights of capability
urights	-User rights of capability
base	-Offset from start of view
limit	-Size of window loaded
money	-Drawing right or money provided by capability
offset	-Logical address of load location
cindex	-Capability index

Description:

Loads a view or part of view provided by a capability into the processes address space.

To nominate the capability index of the loaded capability a non-zero **cindex** should be provided to an empty slot in the table of loaded capabilities. If **cindex** is zero then a value will be automatically allocated.

The kernel can be requested to load a capability at a suitable address to contain the view of the object. The following table gives the values of **offset** and their meanings.

0 load anywhere, preferably a large window
1 load anywhere, preferably a small window
2 load as a large window
3 load as a small window

All other values of **offset** are interpreted as specific addresses. The value of offset is truncated to give a page boundary for small windows or a segment boundary for large windows.

Limit gives the size of the window to be loaded. A **limit** of zero specifies that the limit specified by the capability should be used.

Unload Capability

Name	Symbol	Value
Unload Capability	**K_UNLOADCAP**	**9**

Input Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2
offset -Offset of window to be unloaded
cindex -Index in table of load capabilities of capability to be unloaded

Output Parameters:

limit -Limit of freed window
offset -Offset of freed window
cindex -Index of freed window

Description:

Unloads a capability from address space of the process. If **offset = 0** then the capability **vol serial pass1 pass2** will be unloaded. If **offset = 1** then the capability located at index **cindex** in the table of loaded capabilities will be unloaded. Otherwise the capability at the location **offset** will be unloaded.

Identify Capability

Name	Symbol	Value
Identify Capability	**K_CAPID**	**10**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
offset	-Offset
cindex	-Index in table of load capabilities

Output Parameters:

vol	-Volume of loaded capability
serial	-Serial of loaded capability
pass1	-Password 1 of loaded capability
pass2	-Password 2 of loaded capability
srights	-System rights of loaded capability
urights	-User rights of loaded capability
limit	-Limit of loaded capability
offset	-Offset of loaded capability
cindex	-Index of loaded capability

Description:

Fills in the rights, **limit**, **offset** and **cindex** for a loaded capability. If **offset = 0** then information for the capability **vol serial pass1 pass2** will be returned. If **offset = 1** then the information for the capability located at index **cindex** in the table of loaded capabilities will be returned. Otherwise information for the capability loaded at location **offset** will be returned.

Make Process

Name	Symbol	Value
Make Process	**K_MAKEPROC**	**11**

Input Parameters:

vol	-Volume to create new process on
srights	-Encoded process parameters
urights	-User rights of new process
base	-Start up time for new process
limit	-Highest byte offset of object (hard limit)
money	-Money to be transferred to new process
type	-Type of new process
maxoff	-Maximum offset of new process object (soft limit)
maxsz	-Maximum size of new process object (soft limit)
maxcap	-Maximum number of capabilities for new process (soft limit)
offset	-Offset at which to load new process object
cindex	-Index in table of loaded capabilities for new process object

Output Parameters:

vol	-Master capability (volume)
serial	-Master capability (serial)
pass1	-Master capability (password 1)
pass2	-Master capability (password 2)
urights	-User rights of new process
limit	-Limit of new process object
money	-Money deposited in new process
type	-Type of new process
maxoff	-Maximum offset of new process object
maxsz	-Maximum size of new process object
maxcap	-Maximum number of capabilities for new process
offset	-Offset of new process object
cindex	-Index of new process object in table of loaded capabilities

Description:

Make Process creates an object, loads the object into the current process's address space and fills in the process state information for the new process.

Initially this call creates an object of the size specified on the volume specified with user rights dictated by the **urights** field and system rights set to SRPROCESSMASTER.

Before using the limit value, it is transformed:

$$limit = \begin{cases} BIGLIMIT & \text{if } limit = 0 \\ \text{limit} & \text{otherwise} \end{cases}$$

The new object is created if the following preconditions are met $limit \& 0x3ff \neq 0$, $maxoff \leq limit$, $limit \leq BIGLIMIT$, and $maxsz \leq BIGLIMIT$.

The object is then loaded into the process's address space at either a nominated location or an automatically allocated location. The location is determined by the

value of **offset**. If **offset** is either 0 or 2 then the kernel will allocate a suitable large window automatically and load the object at that location, otherwise the object will be loaded at the segment boundary specified in **offset**.

The capability index of the loaded capability may be nominated by specifying a **cindex** for to an empty slot in the table of loaded capabilities. If **cindex** is zero then a value will be automatically allocated.

A process is then created with the parameters dictated by the **srights** field. The **srights** field is interpreted as four fields of 8 bits:

8 *bits*	8 *bits*	8 *bits*	8 *bits*
Max subp	# message slots	Max loaded caps	# auto load caps

msb lsb

- Max subp - The maximum number of subprocesses for the new process including subprocess 0.
- # message slots - The number of message slots for the new process. As a message slot is reserved for subprocess 0 the number of message slots must be 1 or greater.
- Max loaded caps - The maximum number of loaded capabilities for the new process. This number includes the capability for the process.
- # auto load caps - The number of capabilities to be automatically loaded into the new process's address space including the capability for the new process.

The first four words of the message area contain the initial values of the program counter and stack pointer for subprocess 1. The values are encoded:

message area	index for PC
	initial PC
message area + 2	index for SP
	initial SP

The *index* is the index of a capability in the table of loaded capabilities. If an index value of zero is supplied the initial values are treated as logical addresses instead of as a byte offset from the start of a capability.

The next four words contain the capability of the new process's 'heir'. The heir is notified in case of the death of the process. The message sent contains the remaining cash. If this field contains zero then the master capability for the creating process is used as the heir.

The remainder of the parameter page contains a list of capabilities to be preloaded into the new process's address space. The list is composed of records of the form:

vol	Volume
serial	Serial
pass1	Password 1
pass2	Password 2
base	Start of the loaded window relative to the capability
limit	Size of the loaded window. Zero indicates capability limit
offset	Location of window in the new process's address space
cindex	Index in table of loaded capabilities. Zero for automatic allocation

The creating process will have twice the value indicated in **money** deducted from its cash. This money will be transferred equally to the new process's cash and new process's process object.

The process is scheduled to wake up at the time given in **base** with the wakeup times of **0** and **-1** having the special meanings:

0	Wake up immediately
-1	Set no wakeup time. Awake only when sent a message

Information relating to the new process object is returned to the creating process.

Send Message

Name	Symbol	Value
Send Message	**K_SEND**	**12**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
money	-Amount of money to be sent to process
limit	-Size of message in bytes
offset	-Offset
subpn	-Subprocess number to send message to
cindex	-Index in table of load capabilities

Output Parameters:

srights	-System rights of loaded capability
urights	-User rights of loaded capability
money	-Amount of money sent to process
limit	-Size of message in bytes
offset	-Offset of loaded capability
cindex	-Index of loaded capability

Description:

Sends a message to a process which is loaded into the address space of the sender. If **offset = 0** then the message will be sent to **vol serial pass1 pass2** provided process object is loaded into the sender's address space. If **offset = 1** then the message will be sent to the process with its process object loaded at index **cindex** in the table of loaded capabilities. Otherwise the message will be sent to the process with its process object loaded at location **offset**. The message length is specified in **limit** in bytes. The message to be sent is located at the beginning of the message area. A positive amount of money - **money** - is removed from sender's cash and sent with the message.

Receive Message

Name	Symbol	Value
Receive Message	**K_RECV**	**13**

Input Parameters:

limit -Size of match string

Output Parameters:

money -Amount of money received with message

limit -Size of message

Description:

Recovers message from a subprocess's message queue. If **limit** is non-zero then only a message which matches the first **limit** characters found in the match string will be recovered. The match string is found at the beginning of the message area. The message received is placed into the message area.

If no message is present an error code is returned

External Send Message

Name	Symbol	Value
External Send Message	**K_EXTSEND**	**14**

Input Parameters:

vol -Volume

serial -Serial

pass1 -Password 1

pass2 -Password 2

money -Amount of money to be sent to process

limit -Size of message in bytes

subpn -Subprocess number to send message to

Output Parameters:

money -Amount of money sent to process

limit -Size of message in bytes

Description:

Sends a message to the process **vol serial pass1 pass2**. The message length is specified in **limit** in bytes. The message to be sent is located at the beginning of the message area. A positive amount of money - **money** - is removed from the sender's cash and sent with the message.

External Read Memory

Name	Symbol	Value
External Read Memory	**K_EXTREAD**	**15**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
limit	-Number of bytes to be read
offset	-Offset in bytes from start of capability

Output Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
limit	-Number of bytes read
offset	-Offset in bytes from start of capability

Description:

Reads **limit** bytes from offset **offset** in capability **vol serial pass1 pass2**. The bytes read are stored at the start of the message area.

External Write Memory

Name	Symbol	Value
External Write Memory	**K_EXTWRITE**	**16**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
limit	-Number of bytes to be written
offset	-Offset in bytes from start of capability

Output Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
limit	-Number of bytes written
offset	-Offset in bytes from start of capability

Description:

Writes **limit** bytes from offset **offset** in capability **vol serial pass1 pass2**. The bytes to be written are stored at the start of the message area.

Bank

Name	Symbol	Value
Bank	**K_BANK**	**17**

Input Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2
money -Amount of money to be transferred from capability to cash

Output Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2
srights -System rights of capability
urights -User rights of capability
limit -Size of view in bytes
money -Drawing limit available to capability

Description:

Transfers **money** from cash from the calling process to the capability **vol serial pass1 pass2**. Both positive and negative amounts of cash may be transferred.

If **money** is positive then the capability must have deposit right to perform the transfer. If **money** is negative then the capability must have withdraw right to perform the transfer.

Restrict Rights

Name	Symbol	Value
Restrict Rights	**K_RESTRICT**	**18**

Input Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
srights	-System rights mask
urights	-User rights mask

Output Parameters:

vol	-Volume
serial	-Serial
pass1	-Password 1
pass2	-Password 2
srights	-System rights of capability
urights	-User rights of capability

Description:

Reduces the rights of a capability by performing a bitwise and of the rights masks supplied with the rights bitmaps of the capability **vol serial pass1 pass2**.

The capability named must have suicide right for restrict to operate.

Capability Status

Name	Symbol	Value
Capability Status	**K_CAPSTAT**	**19**

Input Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2

Output Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2
srights -System rights of capability
urights -User rights of capability
base -Cleared by call
limit -Limit of view of capability
money -Withdrawal right of capability
type -Type of object
maxoff -Maximum offset of object
maxsz -Maximum size of object
maxcap -Maximum number of capabilities for object

Description:

Returns details of capability **vol serial pass1 pass2** and associated object.

Rename Capability

Name	Symbol	Value
Rename Capability	**K_RENAME**	**20**

Input Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2

Output Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2
base -Cleared by call

Description:

Changes the passwords of capability **vol serial pass1 pass2** to a new pair of random values.

A precondition to this call is that the capability has suicide right. In addition the master capability of a process cannot be renamed.

Make Subprocess

Name	Symbol	Value
Make Subprocess	**K_MAKESUBP**	**21**

Input Parameters:

base -Start up time for new subprocess
limit -Priority of new subprocess
subpn -Subprocess number

Output Parameters:

base -Start up time for new subprocess
limit -Priority of new subprocess
subpn -Subprocess number of new subprocess

Description:

Creates a new subprocess of the current process. If **subpn** is not zero and no subprocess of the current process has been allocated that number then the subprocess's number will be **subpn**. The priority is set to the least 8 bits of **limit**. The subprocess is scheduled to wake up at the time given in **base** with the wakeup times of **0** and **-1** having the special meanings:

0 Wake up immediately
-1 Set no wakeup time. Awake only when sent a message

The first four words of the message area contain the initial values of the program counter and stack pointer for the new subprocess. The values are encoded:

message area	index for PC
	initial PC
message area + 2	index for SP
	initial SP

The *index* is the index of a capability in the table of loaded capabilities. If an index value of zero is supplied the initial values are treated as logical addresses instead of as a byte offset from the start of a capability.

Delete Subprocess

Name	Symbol	Value
Delete Subprocess	**K_DELSUBP**	**22**

Input Parameters:

subpn -Subprocess number

Output Parameters:

subpn -Subprocess number of deleted subprocess

Description:

Deletes subprocess **subpn**. Note that neither subprocess 0 nor 1 can be deleted.

Load Register Set

Name	Symbol	Value
Load Register Set	**K_LOADREG**	**23**

Input Parameters:

subpn -Subprocess number

Output Parameters:

subpn -Subprocess number

Description:

Copies the structure sysstate at the start of the message area into subprocess table entry **subpn**.

Save Register Set

Name	Symbol	Value
Save Register Set	**K_SAVEREG**	**24**

Input Parameters:

subpn -Subprocess number

Output Parameters:

subpn -Subprocess number

Description:

Copies the structure sysstate from subprocess table entry **subpn** into the start of the message area.

Set Trap

Name	Symbol	Value
Set Trap	**K_SETTRAP**	**25**

Input Parameters:

offset -Subprocess number to send trap message to
subpn -Subprocess number whose trap is being set

Output Parameters:

offset -Subprocess number to send trap message to
subpn -Subprocess number whose trap is being set

Description:

Sets the destination subprocess for trap messages. Subprocess **offset** is notified of faults in subprocess **subpn**.

Receive Message and Close Box

Name	Symbol	Value
Receive Message Close	**K_RECV_CLOSE**	**26**

Input Parameters:

limit -Size of match string

Output Parameters:

money -Amount of money received with message
limit -Size of message

Description:

Recovers message from a subprocess's message queue and closes the mail box the message is recovered from. If **limit** is non-zero then only a message which matches the first **limit** characters found in the match string will be recovered. The match string is found at the beginning of the message area. The message received is placed into the message area.

If no message is present an error code is returned.

Accept Mail

Name	Symbol	Value
Accept Mail	**K_ACCEPT_MAIL**	**27**

Input Parameters:

limit -Size of match string

subpn -subprocess for which mail box is reserved

Output Parameters:

Description:

Opens a mail box for a subprocess and sets the acceptance string for the mail box. The mail box is taken from the pool of closed mail boxes and set to receive messages for a specific subprocess **subpn** or if **subpn** is `0xFF` the mail box can be used for any subprocess.

If **limit** is non-zero then the mail box created will only accept messages which match the first **limit** characters found in the match string when the mail box is opened. The match string is found at the beginning of the message area.

Close Mail Box

Name	Symbol	Value
Close Matching Mail Boxes	**K_CLOSE_BOX**	**28**

Input Parameters:

limit -Size of match string

subpn -subprocess for which mail box is reserved

Output Parameters:

base -Number of mail boxes closed by operation

Description:

Closes mail boxes which match the closing criteria. If **subpn** equals `0xFF` and **limit** is zero then all user mail boxes will be closed. If **limit** is non-zero then only user mail boxes with match strings matching the first **limit** characters of the match string found at the beginning of the message area will be closed. If **subpn** is non-zero then only user mail boxes for subprocess **subpn** are closed.

Copy Object

Name	Symbol	Value
Copy Object	**K_COPYOBJ**	**29**

Input Parameters:

vol	-Volume (original)
serial	-Serial (original)
pass1	-Password 1 (original)
pass2	-Password 2 (original)
srights	-System rights mask
urights	-User rights mask
base	-Start of copy relative to beginning of original
limit	-End of copy relative to base
money	-Money to be transferred to copy
type	-Type of copy
maxsz	-Maximum size of copy
maxcap	-Maximum number of capabilities of copy

Output Parameters:

vol	-Volume (copy)
serial	-Serial (copy)
pass1	-Password 1 (copy)
pass2	-Password 2 (copy)
srights	-System rights of copy
maxoff	-Maximum offset of copy
maxcap	-Maximum number of capabilities of copy

Description:

Duplicates an object by creating a new object and copying the contents of the original object to the new object. This call copies only the defined pages of an object and hence produces an exact duplicate of the contents of the section of the object referred to by the capability for the original object. The rights fields allow the rights of the copy to be reduced as the rights mask and the rights fields are combined by a bitwise AND to produce the copy's rights field. The **money** field indicates the amount of money to be transferred from the process cash to the new object. The **maxsz** field specifies the maximum size of the new object. The **type** field specifies the type of the copy. The **base** field specifies the start of the the copy region which extends through to **limit**. If the **limit** and **base** fields are zero then the complete object is copied.

NOTE:

- This call will **not** duplicate processes
- This call corrupts the first four words of the message area

Check Process State

Name	Symbol	Value
Peek Process	**K_PEEK_PROC**	**30**

Input Parameters:

vol -Volume
serial -Serial
pass1 -Password 1
pass2 -Password 2

Output Parameters:

srights -State of process
base -Wakeup time

Description:

Returns the state and wakeup time of a process given a suitable capability (capability must have SRPEEK right). for the process. The wakeup time is returned in **base** and the process state in **srights**. The process state is encoded:

Value	State
-2	No such process
-3	No right to inquire
1	Process normal
2	Process in kernel
3	Process in read fault
4	Process in write fault
5	Process frozen
6	Process in probate
7	Process dead

Set Heir of Process

Name	Symbol	Value
Set Heir	**K_SET_HEIR**	**31**

Input Parameters:

vol -Volume of heir
serial -Serial of heir
pass1 -Password 1 of heir
pass2 -Password 2 of heir

Output Parameters:

Description:

Set the heir of a process to the capability **vol serial pass1 pass2**. The heir of a process receives a process's death message and any remaining cash.

Appendix B

Formal Description of Restrict

The effect of the **restrict** function on the ability to provide data confinement is formally expressed in this appendix. The following notation will be used:

C - Capability

R - Rights Set of a Capability

M - Rights Mask

O - Origin of the subtree

Superscripts identify the location of an element (relative to the origin of the subtree O) in the subtree. For example $C^{O,3,2}$ is the second child of the third child of the capability at the origin of the subtree. Figure B.1 illustrates a subtree of the capability tree using the notation.

The notion of the *depth* of a node is used in the discussion. The formal definition of the depth of a node relative to the origin of the subtree is provided in equations B.1 and B.2.

The depth of the origin is defined to be zero:

$$depth(C^O) \leftarrow 0 \tag{B.1}$$

The depth of other elements is found using the recurrent relationship:

$$depth(C^{O,\ldots,m,n}) \leftarrow depth(C^{O,\ldots,m}) + 1 \tag{B.2}$$

The rights used in this analysis consist of the union of system and user rights (excluding suicide right - see sections 5.1 and 2.1 for the rational of the suicide

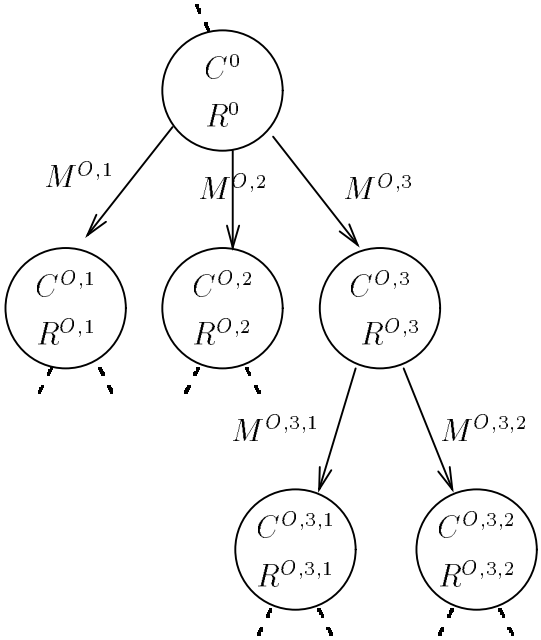


Figure B.1: A Subtree of a Capability Tree

right's exceptional behavior) and the ability to access a range of the contents of the object. Derived capabilities have a subset of the parent capabilities rights and have access to a subrange of the object's address space. Neither of these criteria is strict, so derived capabilities may have equivalent rights and ranges to their parents.

In practice, the selection of pages for a derived capability is performed by using an offset from the base of the parent capability and an extent. The extent is restricted to be less than or equal to the top of the range of the parent's entitlements. For notational convenience the accessible pages of an object will be considered to be a set of access rights, and the rights mask will be extended to represent access to regions of the object. The restriction that the set be contiguous will be implicit throughout this discussion.

When a capability is derived its rights are determined by applying the **logical-and** operator to the rights of the parent capability and the mask:

$$R^{O,\ldots,m} \& M^{O,\ldots,m,n} \rightarrow R^{O,\ldots,m,n}$$

this is equivalent to the set operation:

$$R^{O,\ldots,m} \cap M^{O,\ldots,m,n} \rightarrow R^{O,\ldots,m,n}$$

Thus the following property is true at the time of derivation of a capability:

$$R^{O,\ldots,m,n} \subseteq R^{O,\ldots,m}$$

In the Password-Capability System the relationship is static and hence remains true. The relationship between the rights of a capability ($C^{O,\ldots,m}$) and the rights of its descendents $C^{O,\ldots,m,\ldots,n}$ can be stated:

$$R^{O,\ldots,m,\ldots,n} \subseteq R^{O,\ldots,m} \tag{B.3}$$

The rights of capabilities display the properties of a heap, in that, capabilities at a greater depth than their ancestors are guaranteed to be no more powerful than their ancestors. This property is used to assure users that in giving a capability to another user, the other user cannot use that capability to generate a capability or use the capability in a way which allows the other user to gain access to rights not explicitly conveyed by the capability.

The introduction of the **restrict** operator to the Walnut Kernel invalidates the property defined in equation B.3. To show that the **restrict** operator does not make the system less able to protect users' interests in restricting information flow it is necessary to prove that a new criterion exists of equal or greater strength than the heap criterion of equation B.3.

An enhanced notation is required to handle the description of the **restrict** operation. This notation introduces a subscript to the string used to describe the position of a capability in the subtree of capabilities. The new subscript denotes the number of **restrict** operations that have been applied to a node. Figure B.2 illustrates a subtree of capabilities where the **restrict** operation has been applied to a number of the nodes. The application of the **restrict** operation generates a new tree subtree of the original subtree. The origin of the new subtree overlaps the restricted node. No new nodes can be generated from the restricted node, however, its children are unaffected.

When a capability is restricted its rights are determined by applying the **logical-and** operator to the rights of the original capability and the **restrict mask**. The **restrict mask** is designated by the letter $\mathcal{M}$.

$$R^{O_i,\ldots,m_j} \& \mathcal{M}^{O_i,\ldots,m_{j+1}} \rightarrow R^{O_i,\ldots,m_{j+1}}$$

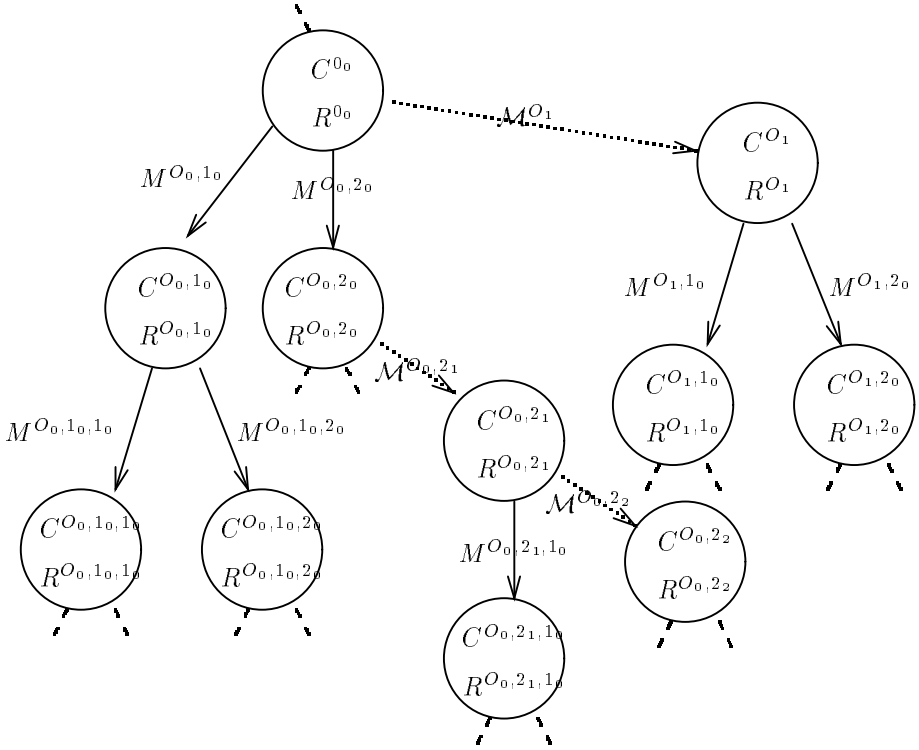


Figure B.2: A Subtree of a Capability Tree - Enhanced Notation

this is equivalent to the set operation:

$$R^{O_i,\ldots,m_j} \cap \mathcal{M}^{O_i,\ldots,m_{j+1}} \rightarrow R^{O_i,\ldots,m_{j+1}} \tag{B.4}$$

As **restrict** may only be applied to the last version of a capability's set of rights, and there is no inverse operation, the following property is true:

$$R^{O_i,\ldots,m_{j+n}} \subseteq R^{O_i,\ldots,m_j}$$

Furthermore, the property

$$R^{O_i,\ldots,m_j,\ldots,n_k} \subseteq R^{O_i,\ldots,m_j} \tag{B.5}$$

is true, as a tree - with the heap property - can be constructed that extends from the origin down to the leaf, by selecting subtrees with origins listed in the path as members of the of the tree, instead of the nodes that represent the latest revisions of the restricted capabilities. Figure B.3 illustrates such a tree derived from figure B.2 for the node $C^{O_0,2_1,1_0}$.

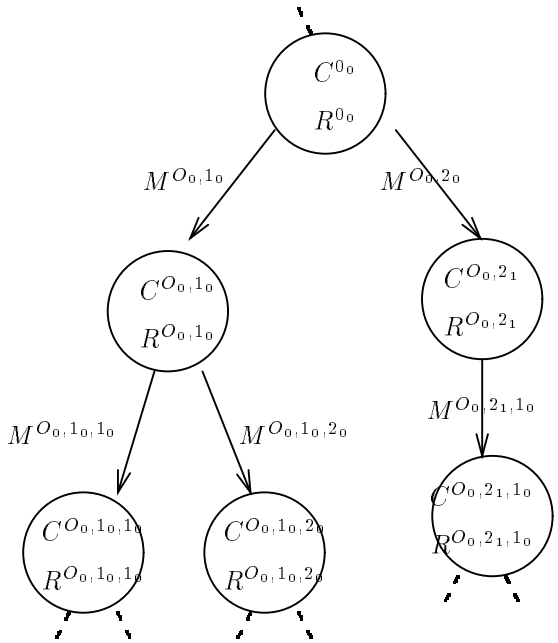


Figure B.3: A Tree with the Heap Property for $C^{O_0,2_1,1_0}$

The presence of such a tree assures users that there has been no loss of security through the introduction of the **restrict** operator. Composing equations B.4 and B.5 provides

$$R^{O_i,\ldots,m_{j+1},n_k} \subseteq R^{O_i,\ldots,m_j,n_l}$$
$$where\ M^{O_i,\ldots,n_k} = M^{O_i,\ldots,n_l}$$

If the restriction at m_j results in a less powerful capability, that is if $R^{O_i,\ldots,m_{j+1}} \subset R^{O_i,\ldots,m_j}$, then the operation may be viewed as a way of trimming the tree of potential branches. The branches eliminated could have held rights in $R^{O_i,\ldots,m_j} \cap \mathcal{M}^{O_i,\ldots,m_{j+1}\prime}$. The ability to trim the tree of potential capabilities enhances the ability of the Walnut Kernel to control access to objects and services.

Appendix C

Hardware Description

This appendix is drawn from a paper presented at ACSC-17 *The Monash Secure RISC Multiprocessor: Multiple Processors Without a Global Clock.* This paper appears as:

Title:	Monash secure RISC multiprocessor: Multiple processors without a global clock
Author:	Maurice D. Castro and Ronald D Pose
Journal:	Australian Computer Science Communications, Proceedings of the Seventeenth Annual Computer Science Conference (ACSC-17) Christchurch, New Zealand
Date:	19-21 January 1994
Editor:	Gupta G
Pages:	453-459

I present it in support of my thesis.

The Monash Secure RISC Multiprocessor: Multiple Processors Without a Global Clock

Maurice Castro* and **Ronald Pose†**
Department of Computer Science
Monash University
Clayton, Vic 3168, Australia
Phone: +61 03 565 5203
Fax: +61 03 565 5146

Abstract

The goal of the Secure Monash RISC Multiprocessor Project is to produce a powerful general purpose scalable multiuser multiprocessor computer. The requirement for synchronous clocks can be a major limitation on both physical layout and electrical design. Instead of attempting to provide a global clock over all processors, we are developing a novel design which has clocks local to each processor and a self clocked bus with asynchronous arbitration. The overall system architecture stresses the ease of scalability by integrating a small switch into the basic processor-memory module, effectively distributing the interconnection network hardware across all nodes.

1 Introduction

During the initial stages of the design of the Monash Secure RISC Multiprocessor it became clear that the existing interconnection networks and bus architectures for multiprocessing computers could not satisfy a number of the design goals of the project adequately. The design required an interconnection structure that was easily scalable, sufficiently versatile to solve general problems, yielded high performance for a large class of problems, and a high degree of fault tolerance.

A novel mesh based interconnection scheme was developed for the bus structure[1]. A consequence of this scheme was a major clock distribution problem: there are multiple paths of differing lengths which must be clocked synchronously. After calculating an approximate size for a processor node board and hence the size of a small, medium and large system, it was apparent that it would be extremely difficult to provide a clock of the required frequency that would be sufficiently well aligned at the processors' bus interfaces to be useful for transferring data across the interprocessor buses.

To overcome these problems a self clocking bus structure was proposed with asynchronous arbitration. This system allows the retention of the design goal of easy scalability and avoids the requirement that the system have a global clock[1]. This permits the construction of a fully distributed system with a passive interconnection scheme.

The use of FIFOs to decouple a bus

*maurice@bruce.cs.monash.edu.au
†rdp@bruce.cs.monash.edu.au

[1]The scheme used for this bus structure will be the subject of a future paper

clock from a processor clock is not unusual[2], however, the use of a deep FIFO capable of holding a number of complete transactions with the aim of preventing the bus being locked by a partially complete transfer is a notable design feature.

2 Design Goals

The design goals of the Monash Secure RISC Multiprocessor project are:

- Scalability - The system will be constructed from modular components enabling the construction of single processor workstations, multiprocessor workstations, medium size multiprocessors and large clustered multiprocessors.
- High Performance - The design will minimize the potential for performance limiting bottlenecks in the bus structure.
- Flexibility - The architecture should be sufficiently flexible to support a variety of algorithms, rather than being tuned to a specific class of algorithms.
- Fault Tolerance - The system should provide for the failure of processors or communications paths in a multiprocessor and provide some means of isolating the faulty component and continuing operation.

3 Design Decisions

Two key design decisions were made at the beginning of the project which have strongly influenced the design:

- Passive interconnection
- Single active module type

All the multiprocessor nodes are interconnected by passive backplanes. Although active interconnections offer a wider scope for avoiding the problems of clock distribution we chose a passive backplane because of its inherent advantages of simplicity and reliability.

A single type of active module offered advantages in the service and design of the machine. In case of failure a module can be unplugged and replaced easily. The design effort is reduced as only a single module needs to be designed.

This arrangement also makes the expansion of the system very simple. Extra modules are purchased and plugged together without requiring any active interconnection components. Unlike currently available massively parallel machines, all the required switching logic is integrated into the basic module. There are no centralized unscalable resources such as active interconnection networks used in other massively parallel machines.

4 Satisfying the Design Criteria

The design proposed for this project attempts to satisfy the project goals by using an interconnection scheme which incorporates both the features of a bus sharing network and a switched network.

4.1 Design Criteria

Scalability

To achieve scalability in small machines a classical SMP (Symmetric Multi-Processor) design is used for communication within a processor board. This gives

[2]The FutureBus+ uses this technique[2]

cost effective scaling for a small number of processors, and allows cache consistency to be achieved using bus snooping.

To build larger systems the processor boards are linked by a mesh-like network. This network is essentially a store and forward network where data is passed from processor board to processor board over a shared external bus. A two part communication protocol is used to ensure that lost messages are accounted for. It is not possible to guarantee cache consistency with this system as only processors on the path of a memory transaction are able to see the contents of that transaction.

Performance

Performance in a multiprocessor system may be limited by several factors. Of most importance to our design were memory bandwidth and memory latency.

The design proposed has high bandwidth, as all connections are made with 64 bit wide buses driven at the processor's external speed. Within processor boards latency is low as the shared internal bus yields a latency proportionate to the speed of the processors and the memory. The latency of interboard communications is higher as transactions crossing multiple processor boards are delayed by the passage through each board.

The latency of external transactions will limit the performance of individual programs, however, as the target operating system is intended to be multiprogrammed, the performance of the overall system will not be limited. By having multiple processors within a node the facilities of the node may be utilized while an individual processor waits for remote memory. When a process is blocked on a memory transaction another process may be allowed to proceed, maintaining system utilization. Furthermore, it is possible for the operating system to relocate processes and their memory pages dynamically so that the path length between processors and data is minimized, hence reducing the overall latency and improving performance.

Flexibility

By adopting a mesh-like external connection it is possible to provide reasonable performance for a variety of algorithms. The system, as it is being implemented, allows for a variety of interconnection topologies with different performance tradeoffs.

Fault Tolerance

A dual ported design is employed which ensures that each processor board has at least two data paths into the external network permitting redundant access to the shared resources. If the external network is sufficiently redundant then it is possible to allow continued operation given a single processor failure. A bus failure can also be tolerated.

4.2 Consequences

To achieve maximum flexibility in determining the configurations of the external network it was necessary to abandon a globally clocked system. This is because we have a large number of small bus segments each with multiple processors. Since each processor spans two buses it seemed natural to have a single clock. However the logistics of distributing such a clock without violating our design aim of a modular system without any critical centralized resources made it necessary to consider alternative arrangements. A self clocked bus was adopted, allowing wide variations in physical path lengths

of the physical buses and eliminating the clock distribution problem[3].

5 Multiprocessor Node

The Multiprocessor Node board (Figure 1) supports a combination of processor or memory modules on the M-P bus (Memory - Processor Bus). A single Multiprocessor Node board behaves as a classic SMP machine. Using the 2 external ports it is possible to connect to an external network of multiprocessor node boards using a passive backplane.

Each Multiprocessor Node Board has a local clock pulse generator. This is used to provide clock signals to the processor and memory daughter boards, the control logic, and the arbiters. This clock is also gated out through the ports to clock the external bus when the port becomes a bus master.

The requirement for a global clock is eliminated by using the FIFOs to decouple the local clocks from the clock found on the external bus.

5.1 Functional Description

The Multiprocessor Node Board consists of 4 major functional blocks connected by a state machine (the control logic). The blocks are:

- M-P Bus
- Bus Switching Unit
- 2 Port Interface Units

M-P Bus

The Memory-Processor bus (M-P bus) is a data, address, and control signal bus. The data and address paths are 64 bits wide, with the data and address signals multiplexed onto the bus. This bus runs using a split bus protocol provided by the processors[4] on the daughter boards plugged into the Multiprocessor Node Board.

The components on the M-P bus form a classical shared memory, SMP machine.

Bus Switching Unit

To help provide off-board communications the switching unit provides 4 operational states:

- Port A connect Port B
- M-P Bus connect Port A
- M-P Bus connect Port B
- No Connection

Port Interface Units

Each port has a port interface unit which performs the 2 functions of transmitting data onto a bus and receiving data from the bus.

To receive data this unit recognizes relevant information on the bus and accepts it into the input FIFO, otherwise bus traffic is ignored.

To transmit data the port interface unit arbitrates for the bus and then outputs data from the output FIFO.

5.2 Operational Description

All addresses in the system are partitioned into 2 regions. The most significant bits of the address determine which Multiprocessor Node Board is to be accessed and the least significant bits determine the address of the memory location on the Multiprocessor Node Board (see Figure 2). Two Multiprocessor Node board numbers are reserved: Node board number zero always refers to memory local to the node board, and the maximum

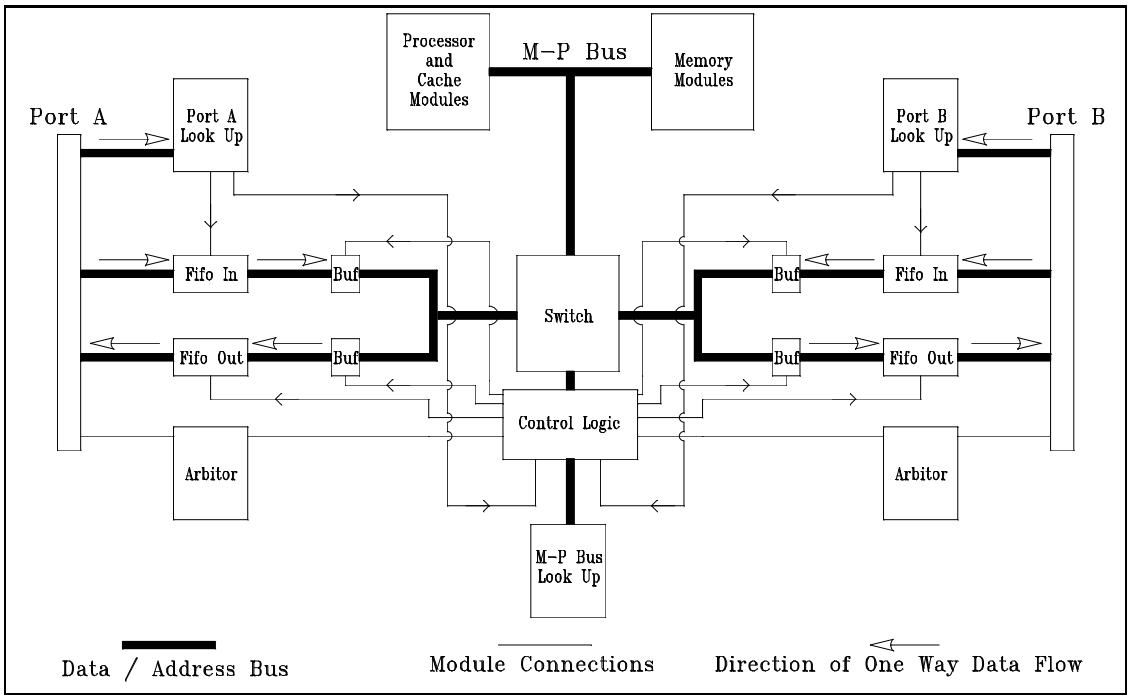


Figure 1: Block Diagram of Multiprocessor Node

node board number refers to hardware control memory local to the node board.

The Node Number is used to index into a routing look-up table held in static RAM, which is decoded to determine where the memory location can be found.

There are three types of access available to the processor:

- Local Memory - Memory is addressed directly over the M-P bus.
- Remote Memory - *Discussed in Section 5.2*
- Hardware Control Memory - The bus ports are isolated and the routing (look-up) tables are modified by the processors.

In addition a Memory to Memory DMA transfer facility will be available to facilitate page sized transfers.

Routing

This section will follow the path of a memory access to illustrate the operation of routing data between memory and processor.

Transfers between nodes employ a packet structure. A packet comprises a header, a body containing the data and a packet check sum. The header contains the source and destination addresses, the packet size and the packet type. Packet types include read, write and an indication of whether the destination is a processor or memory. Packets are constructed and interpreted by control logic in the multiprocessor node board.

Local memory operations use the intrinsic addressing mechanism of the processor.

Memory accesses are routed through the network in a manner similar to a packet based store and forward network.

When a processor utters an off-board address, the high order bits of the address are used to index the M-P Bus look-up table. The look-up table contains bits which indicate which port should be used to attempt the access (Figure 3).

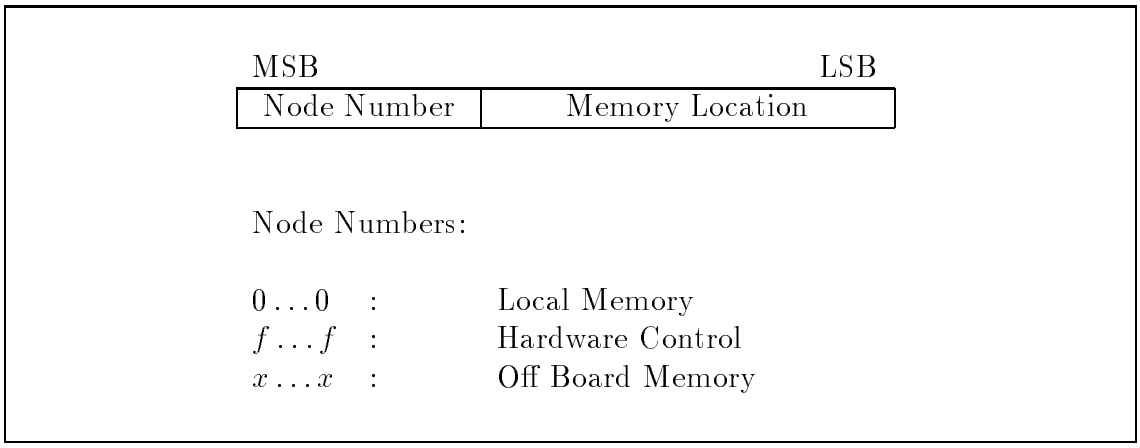


Figure 2: Partitioning of Addresses

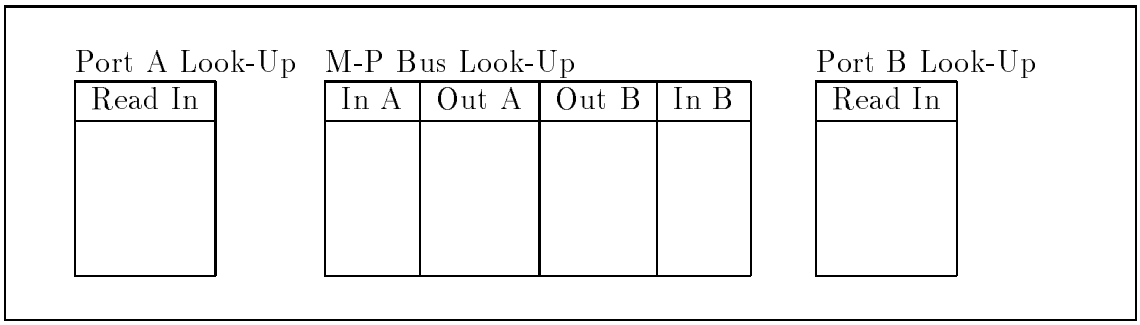


Figure 3: Contents of Look-Up Tables

If the output FIFO on the required port is below the high water mark (the point at which it is guaranteed that the largest permissible packet will fit in the FIFO) and there is no traffic currently passing through the switch, then the switch is connected to the appropriate port. A packet header is constructed and transferred to the FIFO. Data is transferred to the output FIFO. A check sum is added to the FIFO. If the conditions are not met the processor should reattempt the operation later.

When there is data in the output FIFO and there is no bus master an attempt is made to arbitrate for the appropriate bus (The arbiter is discussed in Section 6). When the port becomes the bus master the packet is broadcast onto the bus.

The high order bits of the destination address of the packet on the bus index the port look-up tables of all ports attached to the bus. If the port's 'Read In' bit is set and the input FIFO is below the high water mark then the data on the bus is read into the FIFO, otherwise the data is ignored.

The node number of the destination address in the header of the first packet in the FIFO is used to index the M-P Bus look-up table. If the 'In' bit is set the switch allows the contents of the packet to be directed to the memory on the M-P Bus. Otherwise the switch is set to permit the flow of data from the input FIFO to the opposite output FIFO.

5.3 Design Features

The M-P bus and switched external memory packet transfer allows better uti-

lization of processor memory resources. Both the external and M-P buses may be loaded to the level providing optimal utilization of the bus capacity.

The design introduces a memory hierarchy based on the number of hops between nodes. This feature introduces a new degree of flexibility in the management of both memory pages and processes. The throughput of a processe is maximized by relocation of the process and/or its data to minimize the memory access time. The optimization of overall system performance is complicated by memory being shared by multiple processors. Peak performance is achieved by balancing processor load, memory load, and process average access time.[5].

By employing dual FIFOs on the ports this design attempts to reduce the risk of locking a bus due to traffic from the M-P bus to the other bus port. This design decision adds latency to every transfer through a multiprocessor node, however it significantly increases the bus loading required to cause a bus to be stalled by FIFO being full. This feature is especially valuable where large packet transfers are expected as it effectively doubles the depth of the FIFOs for flow through traffic, hence halving the risk that a packet will not be accepted due to a FIFO being above the high water mark.

6 The Arbiter

Each port uses a priority based arbiter to resolve bus master conflicts. Priorities are rotated to ensure fairness.

The arbiter has the properties:

- Fairness - by rotating the priorities on each attempt to select a bus master each board has an equal chance of being the board with the highest priority in the pool
- Guaranteed Result - A bus master is selected every time an attempt is made.
- Varying Asynchronous Clocks - Arbiters are synchronous with respect to their local clock.

7 Conclusion

The design criteria of easy scalibility and high bus utilization are readily satisfied by the design described. In addition, the multiprocessor nodes, through the use of distributed asynchronous arbitration and a self clocked bus allow a large number of network designs to be evaluated without the need to redesign the multiprocessor nodes. The use of deep FIFOs increases external and internal bus utilization by postponing the onset of bus saturation.

Although a design lacking global clocking was initially considered to be a less attractive option than a globally clocked design we have found that the advantages of dispensing with the global clock (greater flexibility, easier bus design) have overshadowed the cost (higher latency in data transfers).

Acknowledgment

The 'Secure RISC Architecture' project is supported by a grant from the Australian Research Council.

Maurice Castro is a recipient of an Australian Postgraduate Research Award.

References

[1] Patterson D A., Hennessy J L., Computer Architecture A Quantitative Approach, San Mateo, California, 1990, p530

[2] Texas Instruments, Futurebus+ Interface Family Data Manual Preliminary, September 1992

[3] Di Giacomo J., Digital Bus Handbook, New York, 1990, p7.8

[4] MIPS Computer Systems Inc, MIPS R4000 Microprocessor User's Manual, Sunnyvale, California, 1991, p9-3

[5] Bolosky J., Fitzgerald R P., Scott M L., Simple but Effective Techniques for NUMA Memory Management, 12th Symposium on Operating Systems Principles, pp19-31

Appendix D

Glossary

ADC Access Control Descriptors.

AOT Active Object Table.

APD Active Protection Domain.

Copy-on-write A technique which delays the copying of a page until a write to the page occurs. A page is made available to more than one process in a read-only mode, when a process attempts to write to the page the original page is duplicated and the new page is made available to the process to write into.

C-list Capability List.

Copy-on-write A technique which delays the copying of a page until

Capability A number which uniquely identifies an object and access rights to the object or a section of the object.

DMA Direct Memory Access.

DSM Distributed Shared Memory.

FAT File Allocation Table.

IAS Intermediate Address Space.

IPC Interprocess Communication.

LRPC Lightweight Remote Procedure Call.

Mail box A section of the process object used to hold messages

Message area The area of the parameter page that does not contain the parameter block.

Message slot A section of the process object used to hold messages

NUMA Non-Uniform Memory Access. The term is applied to architectures with memory access times varying according to memory location.

OT Object Table.

Password capability A capability composed of two components: a unique identifier for an object and a randomly allocated password.

PDX Protection Domain Extension.

Physical Memory Table This table describes the current state of each page of physical memory.

Physical Object An object on the physical volume which permits access to the first 4 megabytes of the physical memory of the system. This object is used to provide access to buffers used by low level device drivers.

Physical Volume A dummy volume used to contain the physical object.

PMT Physical Memory Table.

Private page tables Second level page tables associated with a specific process. These page tables are used to provide small windows.

Rights A capability has associated with it a set of rights. These rights define the types of access to objects that are conferred by the capability.

RPC Remote Procedure Call.

RPD Regular Protection Domain.

TLC Table of Loaded Capabilities.

Table of Loaded Capabilities A table associated with each process that contains mappings of capabilities to memory locations.

SASA Single Address Space Architecture.

SASOS Single Address Space Operating System.

UI Unique Identifier.

View An attribute of a capability. The region of an object that possession of the capability makes addressable.

Wall A page of memory visible to all Walnut Kernel processes.

Window An area of a process's address space where a view of an object can be loaded.

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and Morrison R. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[AMM+95] T. Agerwala, J. L. Martin, J. H. Mirza, D.C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[And87] Mark Anderson. *A Password Capability System.* PhD thesis, Department of Computer Science, Monash University, 1 1987.

[APW85] M. Anderson, R D. Pose, and C S. Wallace. A password-capability system. Technical Report 52, Department of Computer Science, Monash University, 3 1985.

[APW86] M. Anderson, R D. Pose, and C S. Wallace. A Password-Capability system. *The Computer Journal*, 29(1):1–8, 1 1986.

[ARG89] V. Abrossimov, M. Rozier, and M. Gien. Virtual memory management in Chorus. Technical Report CS-TR-89-30, Chorus systèmes, 4 1989.

[AVW93] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang.* Prentice-Hall, Hemel Hempstead, Hertfordshire, 1993.

[AW85] M. Anderson and C S. Wallace. Security management in a password-capability system. Technical Report 56, Department of Computer Science, Monash University, 8 1985.

[BFF+92] Alan C. Bromberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathon S. Schapiro.

The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112. USENIX Association, 4 1992.

[BFS89] W J. Bolosky, R P. Fitzgerald, and M L. Scott. Simple but effective techniques for numa memory management. In *PROC of the Twelfth SOSP*, pages 19–31, Litchfield Park, AZ, 12 1989.

[BGJ+92] David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randal W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30. USENIX Association, 4 1992.

[BS90] Peter A. Buhr and Richard A. Stroobosscher. The μsystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Software-Practice and Experience*, 20(9):929–964, 9 1990.

[CAC84] W. P. Cockshott, M. P. Atkinson, and K. J. Chisholm. Persistent object management system. *Software Practice and Experience*, 14:49–71, 1984.

[Cas95] Maurice Castro. The Walnut Kernel: User level programmer's guide. Technical Report 95/222, Department of Computer Science, Monash University, 5 1995. revised November 1995.

[Cat88] David Cathro. An i/o subsystem for a multiprocessor. Master's thesis, Department of Computer Science, Monash University, 1988.

[CG87] John Crawford and Patrick Gelsinger. *Programming the 80386*. Sybex, Alameda, Calafornia, 1987.

[Chi90] Vernon L. Chi. Salphastic distribution of clock signals. Technical Report 90-026, Microelectronic Systems Laboratory, CB#3175, Sitterson Hall, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 6 1990.

[CLFL94] Jefferey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single address space operating system. Technical Report Technical Report 93-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, USA, April 1993 (Revised January 1994).

[Cor86] Digital Equipment Corporation. *VAX Architecture Handbook*. Digital Equipment Corporation, 1986.

[Cor90] Microsoft Corporation. QBasic Nibbles, 1990. Source code in BASIC.

[CP94] Maurice D. Castro and Ronald D. Pose. Monash secure risc multiprocessor: Multiple processors without a global clock. In Gupta G, editor, *Australian Computer Science Communications, Proceedings of the Seventeenth Annual Computer Science Conference (ACSC-17) Christchurch, New Zealand*, pages 453–459, 1994.

[CPW95] Maurice Castro, Glen Pringle, and Chris Wallace. The Walnut Kernel: Program implementation under the Walnut Kernel. Technical Report 95/230, Department of Computer Science, Monash University, 8 1995. Also released by SERC, CITRI as Technical Report SERC-0011.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[DdBF+94a] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, David Hulse, Anders Linström, Stephen Norris, John Rosenberg, and Francis Vaughan. Protection in the Grasshopper operating system. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, 9 1994.

[DdBF+94b] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Linström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994.

[DG90] Joseph. Di Giacomo, editor. *Digital Bus Handbook*. McGraw-Hill, New YorK, 1990.

[DLR95] Alan Dearle, Anders Linström, and John Rosenberg. The grand unified theory of address space. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 66–71, 5 1995.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 3 1966.

[Fab74] R S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–412, 7 1974.

[FPR95] Vincent J. Fazio, Ronald D. Pose, and Wells John R. Monash secure risc multiprocessor: Performance simulation. In Kotagiri R, editor, *Australian Computer Science Communications, Proceedings of the Eighteenth Annual Computer Science Conference (ACSC'95) Glenelg, South Australia*, pages 161–165, 1995.

[GC94] Berny Goodheart and James Cox. *The magic garden explained : the internals of UNIX System V release 4, an open-systems design.* Prentice Hall, New York, 1994.

[Geh82] Edward F Gehringer. MONADS: a computer architecture to support software engineering. Technical Report Monads Report No. 12, Department of Computer Science, Monash University, 9 1982.

[Gie90] Michel Gien. Micro-kernel architecture - key to modern systems design. Technical Report CS-TR-90-42, Chorus systèmes, 11 1990.

[GL79] Virgil D. Gligor and Bruce G. Lindsay. Object migration and authentication. *IEEE Transactions on Software Engineering*, SE-5(6):607–611, 11 1979.

[Har85] Norman Hardy. KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, 10 1985.

[HERV94] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating sys-

tem. In G. Gupta, editor, *Proceedings of the Seventeenth Australian Computer Science Conference*, pages 271–280, 1 1994.

[Hil92] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–123. USENIX Association, 4 1992.

[Int90] Intel. *i486 Processor Programmer's Reference Manual*. Intel Corporation, Santa Clara, Calafornia, 1990.

[JJ91a] William Fredrick Jolitz and Lynne Greer Jolitz. Porting UNIX to the 386: A practical approach. *Dr. Dobb's Journal of Software Tools*, pages 16–46, 1 1991.

[JJ91b] William Fredrick Jolitz and Lynne Greer Jolitz. Porting UNIX to the 386: A stripped-down kernel. *Dr. Dobb's Journal of Software Tools*, pages 32–40, 84, 7 1991.

[JJ91c] William Fredrick Jolitz and Lynne Greer Jolitz. Porting UNIX to the 386: The basic kernel. *Dr. Dobb's Journal of Software Tools*, pages 44–56, 8 1991.

[KB] Ted Kehl and Steve Burns. A self-tuned stoppable clock oscillator. Department of Computer Science and Engineering, University of Washington, Seattle, Washington 98195.

[Kee79] J. L. Keedy. A comparison of two process structuring models. Technical Report Monads Report 4, Department of Computer Science, Monash University, 1979.

[Kee82] Leslie J. Keedy. The Monads view of software modules. In A J H. Sale and G. Hawthorne, editors, *Proceedings of the Ninth Australian Computer Science Conference*, 8 1982.

[LDdB+94] Anders Linström, Alan Dearle, Rex di Bona, J. Mathew Farrow, Frans Henskens, John Rosenberg, and Francis Vaughan. A model for user-level memory management in a distributed, persistent environment. In

G. Gupta, editor, *Proceedings of the Seventeenth Australian Computer Science Conference*, 1 1994.

[LMKQ90] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1990.

[Loe92] Keith Loepere. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University, 2.2 edition, 6 1992.

[Mac84] International Business Machines. *Personal Computer Hardware Reference Library: Technical Reference*. IBM, Boca Raton, Florida, first edition, 1984.

[Mic90] Sun Microsystems. *System Services Overview*, revision a edition, 3 1990.

[MIP91] MIPS Computer Systems Inc, Sunnyvale, California. *MIPS R4000 Microprocessor User's Manual*, 1991.

[MSWK93] K. Murray, T. Stiemerling, T. Wilkinson, and P. Kelly. Angel: Resource unification in a 64-bit micro-kernel. Technical Report TCU/SARC/1993/4, City U CS (London), 1993.

[MvRT+90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, pages 44–53, 5 1990.

[MWO+93] Kevin Murray, Tim Wilkinson, Peter Osmon, Ashley Saulsbury, Tom Stiemerling, and Paul Kelly. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proceedings of the USENIX Symposium on Micro-Kernels and Other Kernel Architectures*, pages 31–43. USENIX Association, 9 1993.

[Nor85] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC*. Microsoft Press, Redmond, Washington, 1985.

[OSW+92] P. E. Osmon, T. Stiemerling, A. Whitcroft, A. Valsamidis, T. Wilkinson, and N. Williams. The Topsy project: a position paper. Technical Report TCU/SARC/1992/6, City U CS (London), 1992.

[PFR94] Ronald D. Pose, Vincent J. Fazio, and Wells John R. An incrementally scalable multiprocessor interconnection network with flexible topology and low-cost distributed switching. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter, Special Issue on Interconnection Networks for High-Performance Computing Systems*, pages 31–36, 1994.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, California, 1990.

[Pos91] Ronald David Pose. *A Capability-Based Tightly-Coupled Multiprocessor.* PhD thesis, Department of Computer Science, Monash University, 1991.

[PPTT92] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey. Plan 9, a distributed system. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 31–37. USENIX Association, 4 1992.

[RA85a] John Rosenberg and David Abramson. The MONADS architecture: Motivation and implementation. In *The First Pan Pacific Computer Conference: Proceedings*, volume 1, pages 410–423, 9 1985.

[RA85b] John Rosenberg and David Abramson. MONADS-PC - a capability-based workstation to support software engineering. In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, volume 1, pages 222–231, 1985.

[RAA+91] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS distributed operating systems. Technical Report CS-TR-90-25, Chorus systèmes, 1991.

[RD93] Sub Ramakrishnan and Larry Dunning. On the complexity of task assignment algorithms. In *IX International Conference on Systems Engineering*, pages 166–170, Las Vegas, NV, 7 1993.

[RJO+89] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, and Michael Jones. Mach: A system software kernel. In *Proceedings of the 34th Computer Science International Conference COMPCON 89*, 2 89.

[Rod92] Thomas Roden. High-resolution timing: A fast, tight timer for PCs. *Dr. Dobb's Journal of Software Tools*, pages 42–48, 110, 9 1992.

[RT74] D M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, pages 365–375, 7 1974.

[Sal74] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 7 1974.

[SC90] Julio Sanchez and Maria P. Canton. *IBM Microcomputers: a Programmer's Handbook*. McGraw-Hill, New York, 1990.

[SHFG95] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[ST89] D. L. Schleicher and R.L. Taylor. System overview of the application system/400. *IBM Systems Journal*, 28(3):360–375, 1989.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Tex92] Texas Instruments. *Futurebus+ Interface Family Data Manual Preliminary*, 9 1992.

[Toy91] Michael Toy. Worm, 1991. Part of Berkeley Unix Distribution, Source code in C.

[Tri92] Walter A. Triebel. *The 80386DX Microprocessor: Hardware, Software, and Interfacing.* Prentice Hall, Englewood Cliffs, New Jersey, first edition, 1992.

[Var94] Peter D Varhol. QNX forges ahead. *Byte*, pages 199–201, 10 1994.

[vRT92] Robbert van Renesse and Andrew S. Tanenbaum. Short overview of amoeba. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 1–10. USENIX Association, 4 1992.

[VSK+90] Francis Vaughan, Tracy Schunke, Bett Koch, Alan Dearle, Chris Marlin, and Chris Barter. A persistent distributed architecture supported by the Mach operating system. In *Proceedings of the Mach Workshop*, pages 123–139. USENIX Association, 10 1990.

[Wal94] Walnut Creek CDROM. Freebsd version 1.1. CD-ROM Rock Ridge Format, 5 1994. Walnut Creek CDROM, 1547 Palos Verdes Mall, Suite 260, Walnut Creek CA 94596, USA.