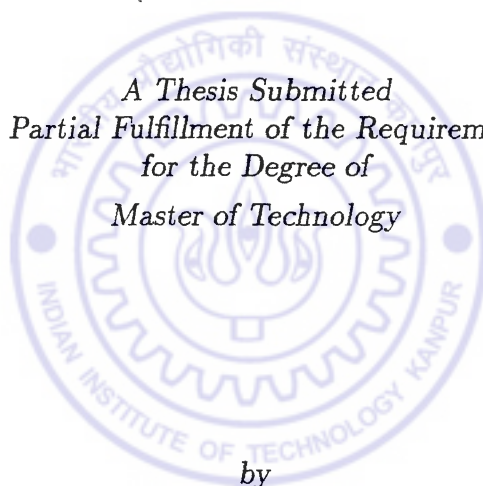


A Generic File System Interface for Embedded Kernels

*A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology*



by
R. Ganesan

to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
April, 1996

28 JUN 1996

CENTRAL LIBRARY
IIT KANPUR

Acc. No. A. 121741



CSE - 1996 - M. GAN - 257

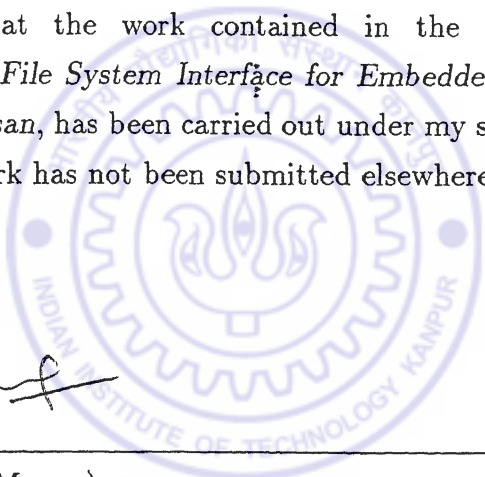
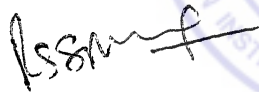


A121741

4/3/96
100 p6

Certificate

Certified that the work contained in the thesis entitled
“*A Generic File System Interface for Embedded Kernels*”, by
Mr. *R. Ganesan*, has been carried out under my supervision and
that this work has not been submitted elsewhere for a degree.



(Dr. Rajat Moona)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

April, 1996

Abstract

Embedded systems have been usually built around proprietary operating systems and interfaces. Even when the external interfaces to these proprietary operating systems are standardized, there is usually no support for subsystems like a file system. Traditionally, embedded systems have found no need for a file system and have handled data storage and manipulation in their own non-standard ways. This thesis discusses the need for a file system interface for object management and emphasizes the necessity of a generic, efficient and light-weight interface. An object oriented design which addresses the issues is then presented. The advantages of using a microkernel based operating system like *Mach* as a basis for designing the interface and developing a prototype implementation is also discussed. Finally a prototype implementation on the Mach microkernel is presented.

Acknowledgments

I would like to thank **Dr. Rajat Moona** for encouraging me to try out something new on *Mach*. His enthusiasm in everything he does is amazing and infectious. I would like to thank **Dr. Gautam Barua** for first introducing me to *Mach*. I wish to thank all the programmers in the computing community who make their work available for free. Their contribution to this field has been enormous and makes computing a pleasure. My friends in *IIT/K* deserve a special mention, they have been excellent company and we've had lots of fun together. I wish to thank my family for encouraging me to do M.Tech. And last, but most certainly not the least, I wish to thank my friend **T.S. Varadarajan** studying in Canada who has been immensely helpful in everything and was instrumental in effectively using the awkward *Internet* policy in this institute.

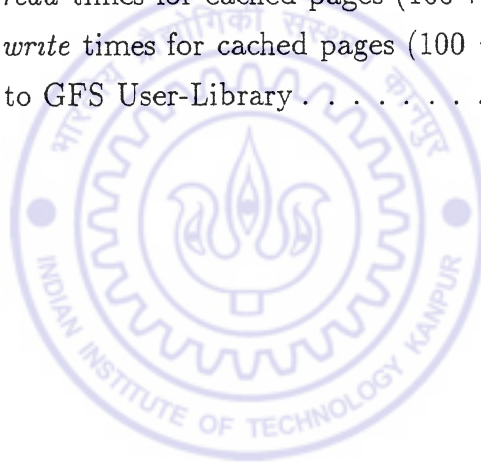
Contents

1	Introduction	1
1.1	Embedded Systems	1
1.2	The Need for a File System	1
1.3	Existing Solutions	2
1.4	Design Issues	3
1.5	File System Interfaces	3
1.6	Mach as a Development Platform	5
1.7	Organization of the Report	6
2	Mach and User-Level Pagers	7
2.1	Introduction	7
2.2	Design Goals and Chief Features	7
2.3	Basic Abstractions	9
2.4	User-Level Memory Managers	11
3	Overview	14
3.1	Overall Architecture	14
3.2	The Embedded File System	15
3.3	Memory Mapped Interface	15
3.4	Interface Design	17
4	<i>Filesys</i> Interface	18
4.1	Introduction	18
4.2	File System Volumes	18

4.3	File System Objects	20
4.3.1	Generic Object	20
4.3.2	Generic File	20
4.3.3	Generic Directory	21
5	User-Library Interface	23
5.1	Introduction	23
5.2	Problems with the <i>stdio</i> Library	23
5.3	Design of User-Library Interface	24
5.3.1	RawFile	24
5.3.2	File	25
5.3.3	Directory	26
5.4	Interaction with the Low-Level Interface	26
6	Implementation	27
6.1	Mach Implementation	27
6.2	Low-Level and User-Library Interfaces	27
6.3	<i>FileServer</i>	28
6.4	Pseudo File System	28
6.5	Benchmarks	28
6.5.1	<i>open</i> and <i>close</i>	30
6.5.2	Uncached <i>read</i>	30
6.5.3	Cached <i>read/write</i>	31
6.6	User-Library Overheads	33
7	Conclusion	35
7.1	Analysis of the Design	35
7.2	Future Work	35

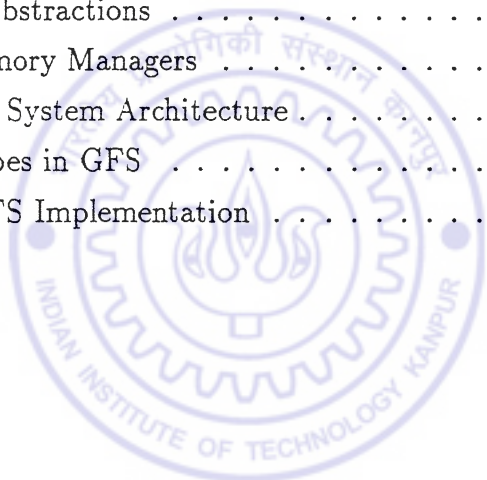
List of Tables

1	Comparison of <i>open</i> and <i>close</i> times	30
2	Comparison of <i>read</i> times for 100 uncached pages	31
3	Comparison of <i>read</i> times for cached pages (100 <i>reads</i> each)	32
4	Comparison of <i>write</i> times for cached pages (100 <i>writes</i> each)	33
5	Overheads due to GFS User-Library	34



List of Figures

1	File System Interfaces	4
2	Mach 3 Structure	8
3	Mach's Basic Abstractions	10
4	User-Level Memory Managers	12
5	Embedded File System Architecture	16
6	Basic Data Types in GFS	19
7	Mach based GFS Implementation	29



Chapter 1

Introduction

1.1 Embedded Systems

Embedded systems encompass a broad range of systems in which computers are used for specialized applications as opposed to general purpose computing. Embedded systems usually use specialized processors such as embedded microcontrollers; many embedded microcontrollers are modified versions of general purpose microprocessors. The diverse applications of embedded systems range from simple controllers for house hold appliances to massive control systems for industrial applications. Typical functionality of embedded systems include data acquisition and processing, command handling from remote systems, storage of acquired parameters for later analysis, process control based on the acquired data etc. *Embedded kernel* or *embedded OS* is the operating system targeted to run on embedded systems.

1.2 The Need for a File System

An embedded kernel also needs to store and manipulate data objects as any other operating system. In a generic OS, the file system provides these services. However a file system is not usually built into a embedded system because space and efficiency are at premium. This usually results in ad-hoc methods of managing memory to be able to accommodate data objects.

A file system is useful in many embedded systems and can swiftly be substituted for non-standard data handling; for example an embedded kernel in a remote data acquisition equipment can use a file system to store it's data temporarily before forwarding it to the final destination; a Cockpit data recorder needs to continuously store the collected data. A file system interface is also useful as a general way of storing and manipulating system objects. Consider a DSP application for example; on-chip memory is at a premium, so a file system can hold the programs or parts of programs that can be overlaid in this fast on-chip memory as and when needed.

There are two reasons why file systems are not used in embedded kernels - one, there is usually no disk or other kind of storage associated with the system and two, a file system could be a unnecessary baggage when memory is at a premium. However as mentioned earlier a file system like interface is useful as a uniform way of manipulating system objects. This thesis provides a solution to both the issues. An interface which serves as a generic interface for managing system objects, which is at the same time light weight and efficient will be presented in the following chapters.

1.3 Existing Solutions

There are many different embedded kernels available in the market, most are proprietary in nature. *QNX*, a microkernel based embedded and scalable Operating System, is one of the few non-proprietary kernels which has been used in a number of embedded applications [Hil92]. Solutions for managing named objects in such kernels vary widely. *QNX*, for example provides a full *POSIX.1* [Lew91] compatible file system for development environments, a simplified file system for embedded applications and different file systems for other media, like a *Flash Memory File System*. Many other embedded kernels provide no support for a file system at all.

It must be emphasized that a complete *POSIX* compatible file system interface is neither needed nor easy to provide in an embedded system because the code complexity will be too high. Therefore, a simplified interface which can be implemented cheaply is needed.

1.4 Design Issues

As the usefulness of a file system depends on the requirements of a particular system, the implementation is likely to vary widely. For example the secondary storage medium may be a disk, flash memory or even part of the main memory. Even for the same medium many different organizations are possible. For example, the file system implementation for a data recorder will be such that appending is very fast. The interface therefore needs to be flexible as well as generic. The other main concern as pointed out earlier is that of space and efficiency. The issues can be summarized as follows

- Well defined implementation independent interfaces should be provided.
- It should be possible for multiple file system implementations to co-exist.
- The interface should allow implementations that are *extremely lightweight*.
- The interface should be *very efficient* - no unnecessary copies from file server or kernel space to user space or buffers.

The first two of these issues are easy to attain, indeed Sun's *Vnode-VFS* interface for Unix achieves exactly this and is used in all modern Unix implementations [Kle86]. The last two issues are the ones we are most concerned with. The overheads needed due to the implementation of file system should be as minimum as possible, otherwise the very purpose of putting a file system where memory is at a premium is lost.

1.5 File System Interfaces

When we talk about a file system interface, there are actually *two* interfaces which come into the picture. The first interface is the interface to the user. This is the *User-Library* interface which is the user's view of the file system. The second interface is the *low-level* interface that is provided by file system itself. This is the concern of the library writer. The OS specific issues in the low-level interface remains hidden

from the end user through a user library that implements the user-library interface. To give an analogy, the user-library interface and the low-level interface correspond to the ANSI *stdio* library built on top of (low-level) system calls in an OS like Unix or DOS.

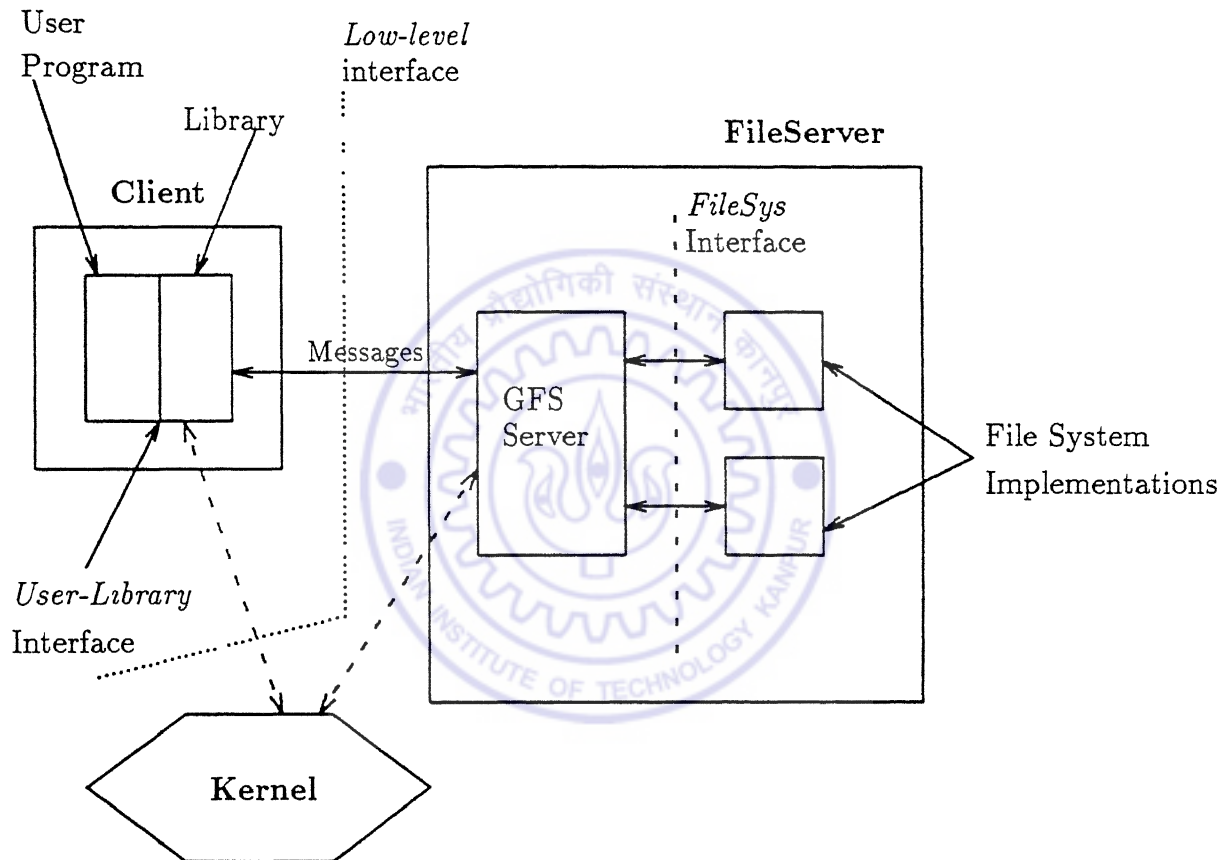


Figure 1: File System Interfaces

Depending on the file system architecture, the interface required by each file system to implement is directly the low-level interface or a *third* interface which we call the *Filesys* interface. The three interfaces are shown in Figure 1 and are discussed in detail in Chapters 3, 4 and 5.

All the interfaces should be thought out carefully. The ANSI *stdio* library has many drawbacks which makes it unsuitable as a user-library interface

[KV91][KSU92]. Similarly alternatives to the Unix system call interface have to be considered keeping in mind the concerns for space and efficiency for the low-level interface.

1.6 Mach as a Development Platform

Microkernel operating systems like Mach [ABB⁺86] and QNX [Hil92] offer several advantages both to OS developers and OS users. Microkernel technology is rapidly maturing and is being increasingly adopted in operating system design. Modern commercial operating systems like *Windows NT* and *OS/2 Warp*, in addition to many Unix variants are based on microkernel technology. It is likely that in future most operating systems will be microkernel based.

Let us consider the advantages that a microkernel operating system offers to the developer. A traditional operating system allows users to add components to a kernel only if they both understand most of it and have a privileged status within the system. Testing new components requires a much more painful edit-compile-debug cycle than testing other programs. It cannot be done while others are using the system. Bugs usually cause fatal system crashes, further disrupting others' use of the system. The entire kernel is usually non-pageable.

A multi-server design [JCS⁺] based on a microkernel like Mach divides the kernel functionality up into logical blocks with well-defined interfaces. Properly done, it is easier to make changes and add functionality. Much more of the system is pageable. Because of clean specifications of the interface, the system can be debugged more easily and new system components can be tested without interfering with other users. But the wall between user and system remains; no user can cross it without special privilege. Since the operating system lies on top of the microkernel, machine dependent part in the OS is minimal which makes it easily portable.

The benefits to the user is obvious. The system is much more stable. The modularity ensures scalability. It is easy to add subsystems or remove ones not needed without bringing down the system and interrupting other users. This modularity and scalability is of primary importance to embedded systems. For example, a file

system need not be included if it is not needed. Considering the success of Mach, QNX and other microkernel operating systems, it is clear that future operating systems will be based on one micro kernel or another. Indeed the success of QNX in the embedded systems market proves that microkernel technology is viable and mature.

Though QNX has proven itself in the embedded market, it is unfortunately a commercial product. The *Mach* microkernel [ABB⁺86] [B⁺93] [Loe91b] developed at the *Carnegie Mellon University* on the other hand is already a mature microkernel which is undergoing further active research in many places, including *University of Utah* and the *OSF Research Institute*. Mach allows user level pagers and provides features like fast IPC, multi-threading and advanced memory management [R⁺88] [Loe91c]. The various versions of Mach are available for free to developers. For these reasons, Mach was chosen as the development platform for the prototype implementation. A brief description of Mach and its features is presented in Chapter 2.

1.7 Organization of the Report

The rest of the report is organized as follows. In Chapter 2 a brief introduction of Mach is presented with particular emphasis to external memory managers. An overview of the thesis is presented in Chapter 3. In Chapter 4 the design of the *FileSys* interface is described in detail, followed by a description of the user-library interface in Chapter 5. An implementation of the GFS interface on Mach is presented in Chapter 6, some benchmark results are also given. We finally conclude in Chapter 7 and give suggestions for further work.

Chapter 2

Mach and User-Level Pagers

2.1 Introduction

The *Mach microkernel* is designed to incorporate many recent innovations in operating system research to produce a fully functional, technically advanced system. Unlike Unix, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. Mach 3.0 microkernel designed at the *Carnegie Mellon University* is a stable and mature platform and serves as a base of many current operating system research projects.

2.2 Design Goals and Chief Features

The key design goals and features of Mach are as follows:

- *Multiprocessor operation:* Mach was designed to execute on a shared memory multiprocessor. Mach provides a multi-threaded model of user processes, with execution environments called *tasks*. *Threads* are pre-emptively scheduled, whether they belong to the same tasks or to different tasks, to allow for parallel execution on a shared memory multiprocessor.
- *Transparent extension to network operation:* Mach has adopted a location-independent communication model involving *ports* as destinations. The original Mach design relies totally on user-level network server processes, though

this model has been changed in recent versions.

- *User-level servers*: Mach supports an object-based model in which resources are managed either by the kernel or by user-level servers. With the exception of some kernel-managed resources, all other resources are accessed uniformly by message passing, irrespective of how they are managed. To every resource, there corresponds a port managed by a server. The *Mach Interface Generator (MiG)* was developed to generate RPC stubs used to hide message-based accesses at the language level [Loe91c].
- *Operating System emulation*: To support the binary-emulation of Unix and other operating systems, Mach allows for transparent redirection of operating system calls to emulation library calls and thence to user-level operating system servers. See Figure 2.

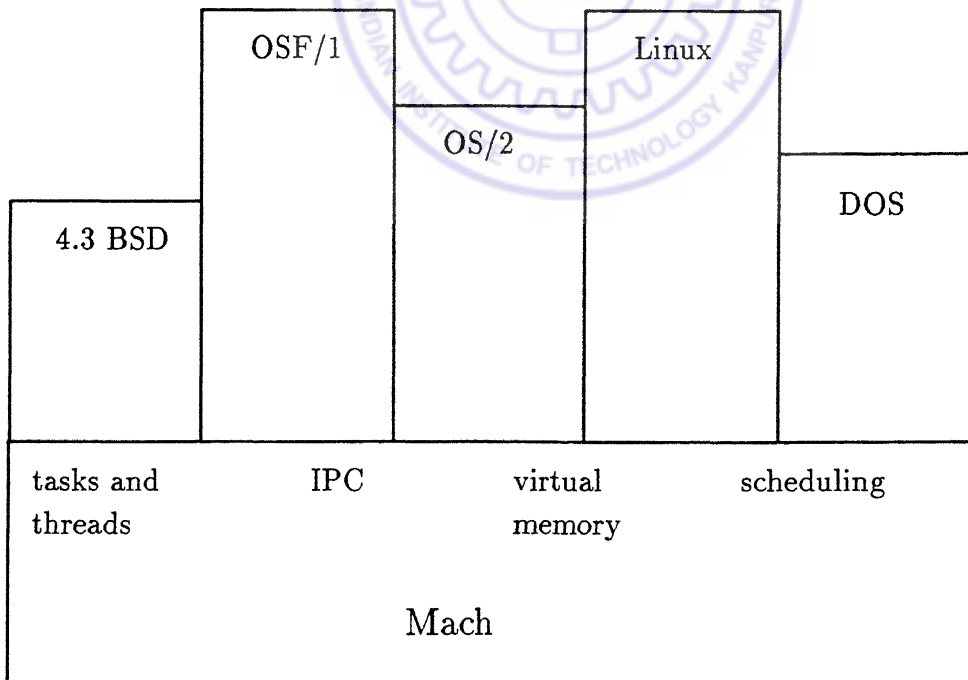


Figure 2: Mach 3 structure

- *Flexible virtual memory implementation:* Mach supports a large, sparse process address space, possibly containing many regions. Both messages and open files, for example, can appear as virtual memory regions. Mach was designed to allow user-level servers to implement backing storage for virtual memory pages. This is a key feature for the *Embedded File System* design and is described in more detail in Section 2.4 below.
- *Portability:* Mach was designed to be portable to a variety of hardware platforms. For this reason, machine-dependent code has been isolated as far as possible. In particular, the virtual memory code has been divided between machine-independent and machine-dependent parts.

2.3 Basic Abstractions

The basic abstractions provided by the Mach kernel are as follows (See Figure 3):

- **Tasks:** A Mach task is an execution environment that provides the basic unit of resource allocation. A task consists of a virtual address space and protected access to system resources via ports. A task may contain one or more threads.
- **Threads:** A thread is a basic unit of execution, and must run in the context of a task (which provides the address space). All threads within a task share the task's resources (ports, memory, and so on). There is no notion of a "process" in Mach. Rather, a traditional *Unix* process would be implemented as a task with a single thread of control.
- **Ports:** A port is the basic object reference mechanism in Mach, and is implemented as a kernel-protected communication channel. Communication is accomplished by sending messages to ports; messages are queued at the destination port if no thread is immediately ready to receive them. Ports are protected by kernel-managed capabilities, or *port rights*; a task must have a port right to send a message to a port. The programmer invokes an operation on an object by *sending* a message to a port associated with the object. The object being represented by a port *receives* the messages.

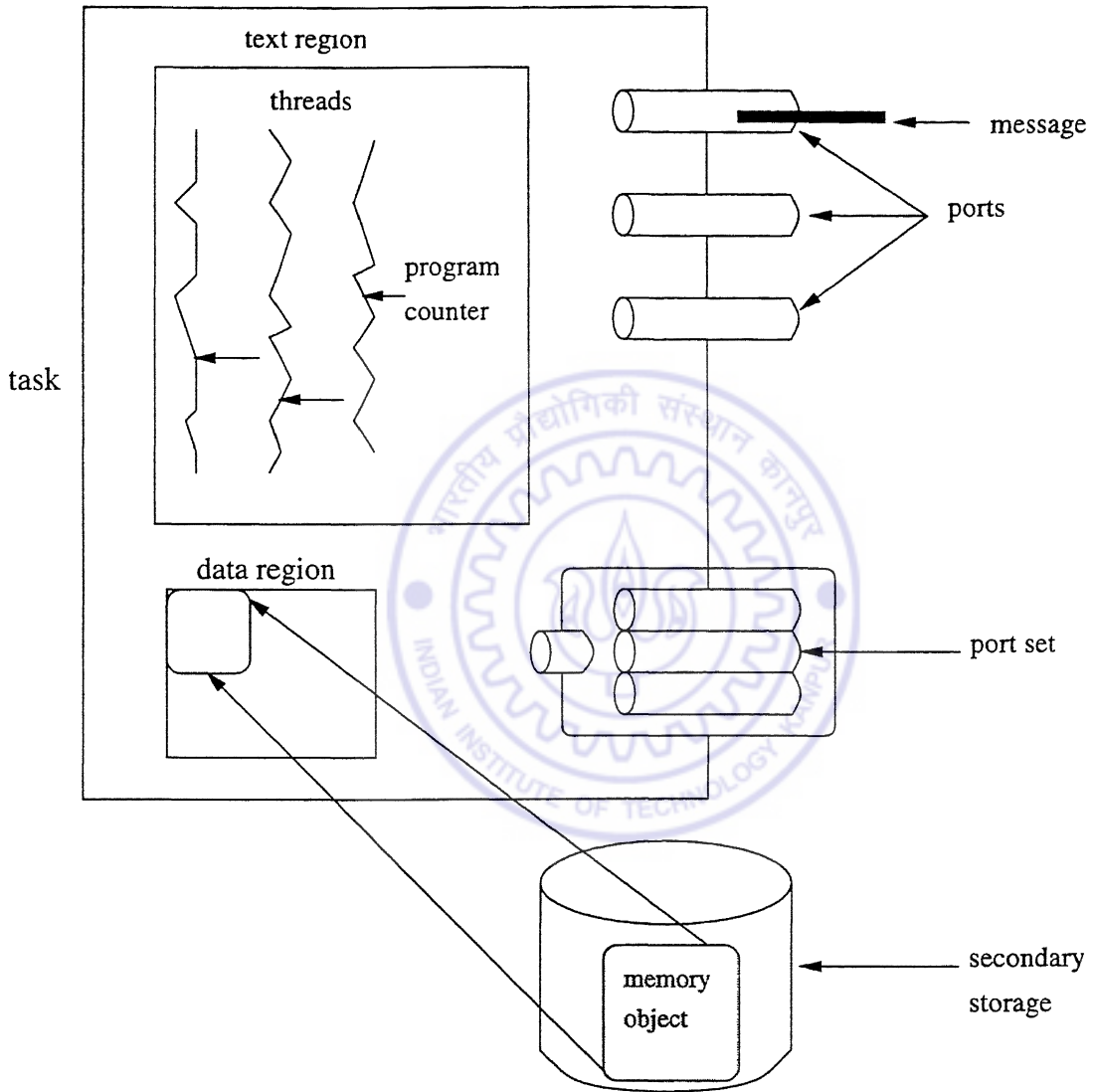


Figure 3: Mach's basic abstractions

- **Port sets:** A port set is a group of ports sharing a common message queue. A thread can receive messages for a port set, and thus service multiple ports. Each received message identifies the individual port (within the set) that it was received from; the receiver can use this to identify the object referred to by the message.
- **Messages:** A message is the basic method of communication between threads in Mach. It is a typed collection of data objects; for each object, it may contain the actual data or a pointer to out-of-line data. Port rights are passed in messages; passing port rights in messages is the only way to move them among tasks.
- **Memory objects:** A memory object is a source of memory; tasks may access it by mapping portions (or the entire object) into their address spaces. The object may be managed by a user-mode *external memory manager*. Memory objects and external memory managers are described in detail in the next section.

2.4 User-Level Memory Managers

User-level memory managers is a key feature of that is essential for this thesis. A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual-memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept of a memory object being created and serviced and maintained by nonkernel tasks is important. This makes user-level memory managers possible. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage.

Basic Manipulation Manipulation of a virtual address space by a user-mode task takes the following basic form:

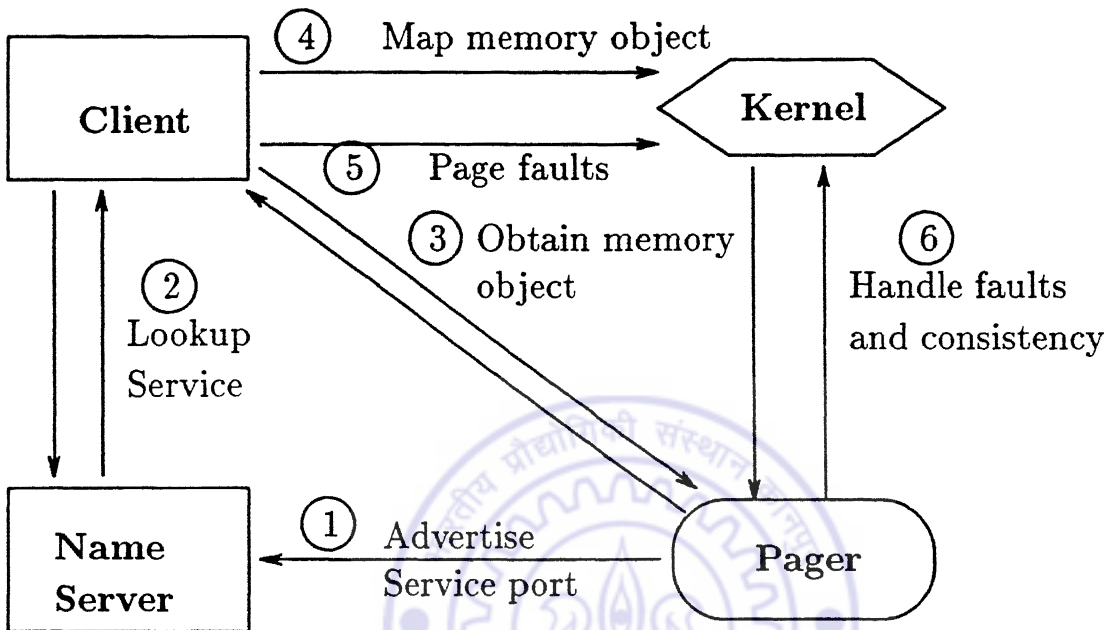


Figure 4: User-level Memory Managers

- A task obtains the port for an object that can be mapped.
- A task establishes a new memory range by invoking the *vm_map* system call. Included in this system call is a port which identifies the object, and the memory manager which is responsible for the region. The kernel executes calls on this port when data are to be read or written in that region.
- The task attempts to reference a portion of this memory range (most likely simply by touching it). Since that portion does not yet exist in memory, the referencing task takes a page not resident fault. The kernel sends a message to the range's abstract memory object requesting the missing data. The reply from the abstract memory object resolves the requesting task's page fault.
- Eventually, the resident pages of the memory range, with values possibly modified by the client tasks, are evicted from memory. Pages are sent in messages to the range's abstract memory object for their disposition.

- The client task de-establishes the memory range by calling the system call *vm_deallocate*. When all mappings of this memory object are gone, the abstract memory object is terminated.

The kernel should be viewed as using main memory as a (directly accessible) cache for the contents of the various memory objects. The portion of this cache that contains resident pages for a memory object is referred to as the *memory cache* object.

Memory Managers The kernel is involved in a dialog with various memory managers to maintain it's memory cache, filling and flushing this cache as the kernel sees fit. The dialog consists, in general, of asynchronous messages, as the kernel cannot be stalled by a memory manager, and memory managers wish the maximum possible concurrency in their operations.

When the first *vm_map* call is made on a memory object, the kernel sends message to the memory manager port passed in the call, invoking the *memory_manager_init* routine, which the memory manager must provide as part of it's support of a memory object. Two ports are passed to the memory manager in this message (and most other messages), called the *control port* and *name port*. Name ports are used throughout Mach. They are used simply as a point of reference and comparison. Finally, the memory object must respond to the *memory_manager_init* call with a *memory_object_set_attributes* call to indicate that it is ready to accept requests. When all tasks with send rights to a memory object relinquish those rights, the kernel deallocates the object's ports, thus freeing the memory manager and memory object for destruction.

There are several kernel calls that are needed to support external memory managers. The memory manager itself must provide support for several calls so that it can support an object. For instance the *memory_object_data_request* call must be provided to handle a page fault in a user-level task which has mapped an object supplied by this server. The various calls are well documented in [Loe91d, Loe91c].

Chapter 3

Overview

3.1 Overall Architecture

The GFS architecture is based on a *client-server* model, keeping in-tune with the microkernel philosophy. With this model, there are two possible designs in the overall architecture of the file system. The first approach is to have a single file system server which implements all the different file systems. The second one is to have a separate server for each mounted file system and (if necessary) a master file server to manage the namespace. The second design is obviously more modular and is probably the best choice in a general purpose OS because of the flexibility it offers.

The main drawback of the multiple server design is that this may result in a lot of duplicated functionality in each file system implementation. Except for low-level I/O and a few other details most file system implementations usually have a lot in common. This code is unnecessarily duplicated in each file server. This is not acceptable in an embedded system. Therefore the first design was chosen. There is another advantage to the first design, the design can be adapted into a monolithic kernel also with few modifications; this is much more difficult in a distributed design.

A design for a single file server does not automatically preclude it's adoption for multiple servers when it is suitable. A careful design of the interfaces and name space management can ease the adoption of the design for multiple servers.

3.2 The Embedded File System

The embedded file system architecture has a single file server, simply called *FileServer* which can have multiple file system implementations built in. Clients which need to access different file systems communicate with *FileServer* by either explicit messages or indirectly through a *memory-mapped* interface (see Section 3.3 below.). The architecture of the system is shown in the Figure 5. *FileServer* exports the *low-level* interface mentioned in Section 1.5. Each file system implementation is not concerned with this low-level interface and exports a standard interface called the *Filesys* interface. This *third* interface is needed because the low-level interface may be dependent on the target operating system and must be insulated from the file system implementor. While three interfaces may seem to be a bit complicated, this design provides the maximum flexibility and is no different from the Unix hierarchy of *VFS-Vnode* interface, the Unix system call interface and the ANSI *stdio* interface. In the case of a microkernel architecture the low-level interface will be *RPCs* from the *client* (user program) to *FileServer*.

File systems are called *Volumes* and are identified by a unique 8-byte *Volume ID*. Each volume is in its own name space; a full pathname begins with a Volume ID, followed by a : (colon), followed by the actual path (similar to MS-DOS). This scheme was chosen over a single hierarchical name space as in Unix because name lookup is much simpler and separating out servers for each volume is easier.

3.3 Memory Mapped Interface

A direct function call interface for all file system related calls including *read* and *write* similar to Unix is an obvious choice for the low-level interface but suffers from two disadvantages; the first one is that there are unnecessary copies from system space to user space and vice versa. In the case of microkernels this copying may be avoided through memory management optimizations but RPC costs for every read/write may be prohibitively expensive [DA92]; the second disadvantage is the need of a cache to maintain acceptable levels of performance. In Unix, the *buffer-cache* is an important part of the file system [Bac86]. While the buffer-cache is

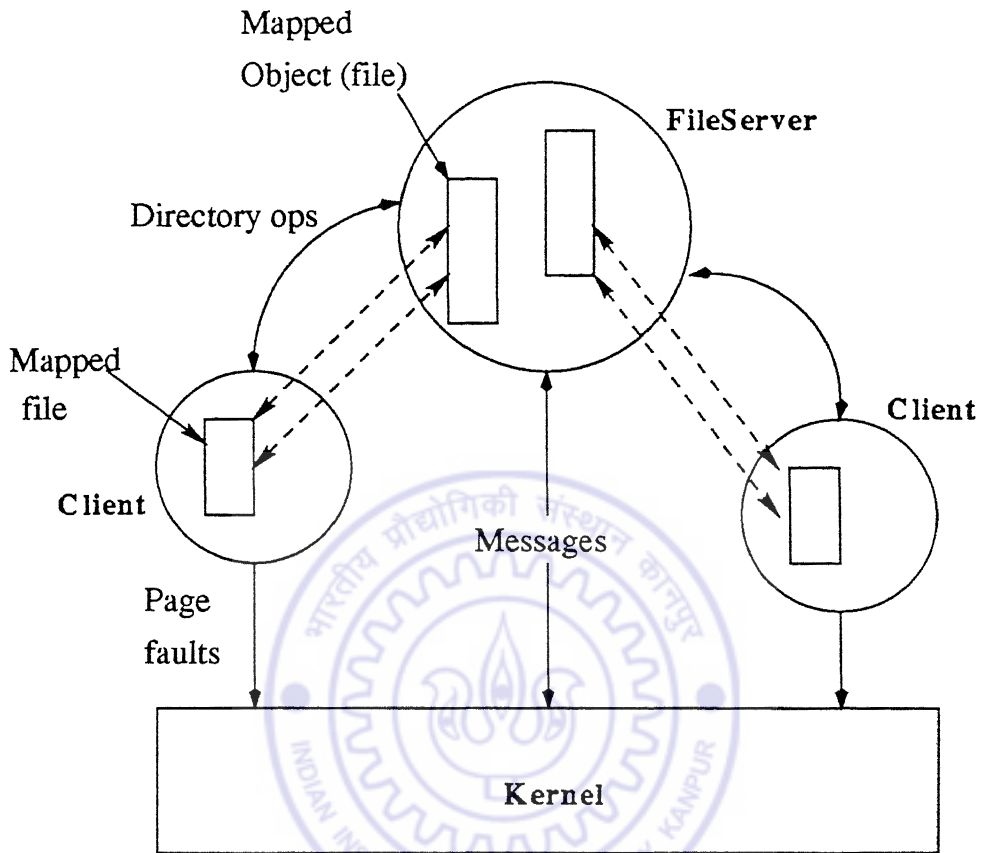


Figure 5: Embedded File System Architecture

efficient and elegant, it also contributes significantly to the code size of the file system.

The overall concern for a light weight interface and efficiency, therefore makes the choice of a memory mapped interface inevitable. *Krieger et al.* [KSU92] show that use of memory mapped files in standard Unix make applications run two to three time faster. As in any case memory mapping will be there for other functions of an embedded controller, this will pose very little overhead too. Moreover, in this approach the pager functionality will be provided through a shared library in systems using user-level memory managers; for example the *GNU Hurd* operating system does this over the Mach microkernel [Sta90].

A mapped file implementation eliminates all unnecessary copies. More importantly *there is no need for a separate buffer cache* as in Unix since the pager functionality automatically takes care of caching. *Gingell et al.* [GMS87] discusses the advantages of memory mapped files and the unified memory cache for all system objects. User level memory management is usually available in embedded micro kernels or should be relatively easy to implement. A mapped interface does not imply that it has to be through paging. Systems without virtual memory can achieve similar results by overlaying a portion of the process' address space. The Mach micro kernel supports user-level pagers directly as described in Chapter 2.

3.4 Interface Design

A memory mapped interface is efficient, but is not very convenient to use. A suitable *user-library* interface has to be provided (See Section 1.5). The low-level interface is operating system specific as mentioned earlier and the Mach implementation is through RPCs and the *vm_map* system call.

An object-oriented design using a hierarchy of C++ classes is presented [ES90] for both the *Filesys* interface and the *user-library* interface. C++ is was chosen for interface specification because it is gaining importance as a system programming language. Moreover, file system interfaces naturally fall into an object oriented class hierarchy and abstract base classes in C++ are good at expressing interfaces [Str91, Str94]. And finally the freely available *GNU C/C++* compiler suite for most architectures makes porting a non-issue. The *FileSys* interface and the user-library interface are described in detail in the following two chapters.

Chapter 4

Filesys Interface

4.1 Introduction

This chapter details the *Filesys* interface, the interface that must be implemented by a new file system implementation. There are primarily two different kinds of objects in a file system; file system volumes and objects in the volumes. The objects in the volumes are files and directories.

The *Filesys* interface is loosely based on Sun's *Vnode/VFS* architecture for Unix [Kle86]. An object residing in a volume (file or directory) is abstractly modeled as a *GObj* (generic object). File and directories are derived from *GObj*. The file and directory are called *GFile* and *GDir* respectively. Similarly each mounted file system is represented by an object called a *GVol* (generic volume). *GObj*, *GFile*, *GDir* and *GVol* are all abstract classes and don't represent real objects. They are instead used to export standard interfaces that a file system implementor has to implement. Since file attributes and directory entries have to be generic, a common base structure for these entries has been defined for this data types. Figure 6 shows these data types.

4.2 File System Volumes

A mounted file system is represented by an object called a *GVol* (generic volume). A new file system implementation derives from *GVol* and implements the required

```

typedef char gvalid_t[8];

typedef struct {
    enum modes { rdonly = 0x0001, rdwr = 0x0002, dir = 0x0100 };

    size_t    ga_length;        /* file size */
    unsigned  ga_mode;         /* file modes */
} gattr_t;

typedef struct {
    unsigned  gd_id;           /* Unique id */
    char      gd_name[256];    /* file name */
} gdirent_t;

typedef struct {
    gvalid_t  sg_volid;        /* name of the volume */
    unsigned  sg_pages;        /* total # of pages */
    unsigned  sg_free;         /* # of free pages */
    unsigned  sg_files;        /* # of files (including dirs) */
} statgvol_t;

```

Figure 6: Basic Data Types in GFS

interface methods. The operations on a *GVol* are:¹

gv_mount(gvalid_t volid)

Mount the file system volume with volume id *valid*. The volume must be of the expected type.

gv_unmount()

Unmount the file system and free it's resources.

gv_root(GObj **gpp)

Return a pointer to the root *GObj*² for this file system in *gpp*.

¹All operations are method invocations on the respective objects, the object itself is implicitly passed in the call.

²*GObj* is a generic object in the file system and is described below.

gv_statvol(statgvol_t *svp)

Return information about a mounted file system. *svp* points to a *statgvol_t* structure for the results. The *statgvol_t* structure is shown in Figure 6

4.3 File System Objects

The *GObj* is the base for the file system objects (files and directories). It is analogous to the *inode* in Unix and keeps track of open objects in the system. The *GObj* exports a minimal set of generic operations common to both files and directories. *GFile* and *GDir* inherit from *GObj* and exports operations on files and directories.

4.3.1 Generic Object

The *GObj* represents data and functions that are common to both files and directories. There are only three operations defined in this class. All operations return error codes indicating success or failure. The operations are:

go_getattr(gattr_t *gap)

Get the attributes of the *GObj*, where *gap* is a pointer to the attributes structure of the object. See Figure 6.

go_setattr(gattr_t *gap)

Set the attributes of the *GObj*.

go_sync()

Write out the cached information for this *GObj*.

4.3.2 Generic File

The generic file object is called *GFile* and inherits from *GObj*. File I/O is through mapped files for efficiency and flexibility. However, a pager interface is by nature operating system specific, therefore *GFile* only requires generic *read* and *write* operations that are page aligned. The pager interface is then an implementation specific issue and calls *read* or *write* in responding to a paging request. Though direct *read* and *write* calls could be supported by the file system, the mapped interface should be preferred. The *GFile* operations are:

gf_open()

Open the file for reading. This method is called by *FileServer* when a client maps in this file.

gf_close()

Close the file, flushing all buffers to backing storage. This method is called when all clients have unmapped this file.

*gf_read(unsigned pageno, unsigned n, void **datapp)*

Read *n* bytes of data starting from page *pageno*. *datapp* points to a pointer to the returned page(s).³

*gf_write(unsigned pageno, unsigned n, void *datap)*

Write *n* bytes of data starting from page *pageno*. *datap* is a pointer to the data to write.

*gf_link(GDir *tgdp, char *tnm)*

Link this object into directory *tgdp* under the name *tnm*. This call may not be implemented if the underlying file system does not understand multiple links to the same file.

4.3.3 Generic Directory

The generic directory object is called *GDir* and inherits from *GObj*. *GDir* exports a uniform interface for directory operations. Though a generic directory structure for operations on directories is defined (See Figure 6), a particular file system is free to implement it's own directory structure internally. The *GDir* generic operations are:

*gd_create(char *nm, gattr_t *gap, GFile **gfp)*

Create a file named *nm* in this directory with attributes *gap*. *gfp* is a pointer to the newly allocated *GFile* that is returned.

*gd_rename(char *nm, GDir *tdgp, char *tnm)*

Move the object named *nm* in this directory into the target directory *tdgp* under the name *tnm*.

³We return a pointer to a pointer to avoid an unnecessary copy here.

28 JUN 1996

*gd_remove(char *nm)*

Remove a file with name *nm* in this directory.

*gd_lookup(char *nm, GObj **gpp, unsigned lookupflags)*

Lookup a pathname component *nm* in this directory. *gpp* points to a pointer to pointer to a *GObj* for results. *lookupflags* indicates whether to lookup for *read/write*.

*gd_mkdir(char *nm, GDir **gdpp)*

Create a directory *nm* in this directory. The newly created directory is returned in the pointer to a *GDir* pointer *gdpp*.

*gd_rmdir(char *nm)*

Remove a directory *nm* from this directory.

*gd_readdir(dirent_t *dirp, int &nentries)*

Read the contents of this directory. The results are returned in an array of *dirent_t* structures, *dirp*. *nentries* is the number of entries in *dirp*

GDir doesn't have explicit *open()* and *close()* operations because a directory is implicitly opened when looked up and closed when nobody refers to it, whereas *GFile* is opened only when it is mapped. The difference is because *GFile* operations are meaningful only when somebody maps in a file unlike *GDir* operations.

Chapter 5

User-Library Interface

5.1 Introduction

The user-library interface for GFS provides fast stream I/O. It is a complete re-implementation of the *Alloc Stream Interface* described by *Krieger et al.* [KSU92]. It is also modified to fit into the object-oriented framework of GFS. The user-library interface is not only faster than the ANSI *stdio* library, it is also very light-weight and *thread safe*. In this chapter the problems of the *stdio* are discussed and the new user-library is discussed in detail.

5.2 Problems with the *stdio* Library

There are two main disadvantages with the Unix I/O interface: Firstly, as the interface now stands, an excessive number of system calls result if a user process accesses a file with many small read and write operations. Secondly, the user supplies a private buffer into which the data should be read, or from which data should be written. This can result in a large performance cost when data is copied to and from the system buffers.

To reduce the number of interactions with the operation system, the *stdio* runtime library buffers data in the application's address space and thus amortizes the cost of interactions with the operating system over several application requests.

However, buffering at the user level introduces yet another layer of copying, namely between library and application buffers. Moreover, most implementations of *stdio* are not re-entrant, which means that in a multi-threaded environment each thread is forced to do explicit locking and unlocking before and after each operation.

The GFS design adopted a mapped interface to reduce the copying between different buffers, and can allow for different threads in an application to concurrently access different parts of the same file. The user-library should be also designed in the same spirit by avoiding unnecessary copies.

5.3 Design of User-Library Interface

The user-library interface for files are modeled after the Unix memory allocation interface (i. e. *malloc*, *realloc*, *free*). The interface returns a pointer directly into library buffers (which is in fact a pointer into the mapped region), so there are no unnecessary copies. Another advantage of the interface is that the user need not learn a totally new interface since it is modeled after the standard *malloc* interface. There are no new directory operations in the interface, traditional interfaces or similar ones are provided.

5.3.1 RawFile

The base object for the user-library interface is called the *RawFile* and exports operations that are common to both files and directories. The operations exported by *RawFile* are as follows:

open(char *path, unsigned flags, unsigned mode)

Open file/directory for reading. The *flags* parameter specifies lookup for read/write/both and the *mode* parameter indicates the creation mode if the file is not found and is opened for write.

close()

Close the file or directory, flushing the contents to secondary storage.

flush()

Flush the contents to secondary storage.

5.3.2 File

The *File* object inherits from *RawFile* and exports the *malloc* kind of interface to access memory-mapped files provided by the *FileServer*. The *File* object maintains a doubly-linked list of mapped regions and exports the following methods:

void *salloc(int &length)

salloc is similar to *malloc* and allocates *length* bytes from the current offset in the file and returns a pointer to the allocated area. *length* is a in/out parameter and returns the actual length allocated or a negative error code if the request could not be satisfied.

void *srealloc(void *bufp, int &length)

srealloc is similar to *realloc* and resizes an already allocated buffer *bufp* and adjusts it to the size *length*. *length* returns the actual size or a negative error code if the request cannot be satisfied.

void *sallocAt(void *bufp, int offset, int &length)

In *Unix* random access to a file is by a *seek* followed by *read* or *write*. In a multi-threaded environment, the file has to be locked before the *seek* and unlocked only after *read* or *write*. To avoid this, *sallocAt* provides an interface which returns a pointer to a buffer of size *length* at the given *offset*.

free(void *bufp)

free() frees a buffer which was allocated earlier. *bufp* should be the a pointer returned by an earlier *salloc*, *sallocAt* or *srealloc*.

The *salloc*, *srealloc*, *sallocAt* and *sfree* methods lock the object before the method is performed for safe multi-threaded operation. There are faster versions named with a prefix *u_*, namely *u_salloc*, *u_srealloc*, *u_sallocAt* and *u_sfree* which do not lock the object. These faster can be used if there is only one thread or when the lock is obtained once before a set of consecutive operations.

5.3.3 Directory

The directory object *Dir* inherits from *RawFile* and its interface is traditional. There would be no benefit in using mapped files for directory operations because the internal directory structure is specific to a file system implementation and a conversion to the generic structure (and hence a copy) is always necessary. The directory operations are as follows:

readdir(dirent_t *entryp)

This method returns a single directory entry in *entryp*. Repeated calls of the function return consecutive entries in *entryp*.

mkdir(char *name)

Create a new directory with name *name*.

rmdir(char *name)

Remove the directory with name *name*. The directory specified by *name* must be empty.

unlink(char *name)

Remove a file with name *name*.

rename(char *oldname, Dir *newdir, char *newname)

Move the entry *oldname* from this directory to the directory *newdir* under the name *newname*.

5.4 Interaction with the Low-Level Interface

The details of how the user-library interface maps on to the *low-level* interface is left unspecified on purpose. The *File* object for example expects some equivalent to the Unix *mmap* call in order to map files. The actual call that it uses is operating system specific. The *Dir* object similarly expects low-level operations corresponding to it's exported interface, in trivial cases these methods may be inline expansions that call the low-level interface without any library overheads. Some issues specific to the Mach implementation is presented in the following chapter.

Chapter 6

Implementation

6.1 Mach Implementation

An implementation of the three interfaces was developed on *mach4-uk02p13*¹ running *Lites-1.1.u3*, a BSD compatible Unix server and emulator distributed by the *University of Utah* [Hel94] hosted on *NetBSD-1.0*. The compiler tools used were from *Free Software Foundation's* GNU C/C++ compiler suite. The entire environment is freely downloadable from the Internet.

6.2 Low-Level and User-Library Interfaces

The low-level interface for most operations is Mach RPCs, these RPC calls are defined using the MiG language specification. The user-library makes these RPC calls on behalf of the client. For mapped files the interface is using the *vm_map* call described in Chapter 2. The user-library interface has been implemented using *vm_map* call and RPC stubs generated using MiG.

¹Mach 4, is currently just Mach 3.0 from CMU in a new, flexible build environment and a few minor modifications.

6.3 *FileServer*

FileServer is a multi-threaded user-level Mach server. *FileServer* has two sets of threads. The first set handles name lookup and volume operations like mounting and unmounting. The second set of threads handle object operations. This set of threads function as external memory managers for file objects and also as simple servers for directory operations. The overall architecture is shown in Figure 7.

FileServer begins its operation by registering its port with a name server with the name *GFS_Server* so that clients can lookup the name and can access it. Then it forks off the threads for volume requests like *lookup*, *mount* etc. This is the first set of threads. Next it allocates a *port set* for object operations (including paging) which is initially empty and forks off threads to service these object requests.

Once a file system is mounted and a object is looked up, a unique *port* is created for the object and moved into the object port set. In the case of directories all directory operations are direct RPC calls to the the object port. File I/O is implemented through the Mach *External Memory Manager* interface, where messages are exchanged with the kernel and not directly with the client. In either case any of the object threads receive a message and invoke the appropriate object method.

6.4 Pseudo File System

A *Pseudo File System* (PFS) which uses a Unix file system provided by the *Lites* server was implemented to test the functionality of the GFS interface. PFS converts all object operations into file system calls on the underlying Unix file/directory. The PFS is only intended to demonstrate that the *FileSys* interfaces are easy to use and efficient. A few benchmarks are presented in the next section for comparison.

6.5 Benchmarks

The benchmarks compare the performance of *FileServer* with PFS on the Mach micro kernel with the Lites server on the same machine and with native NetBSD-1.0. All timings were taken on a 486 DX2 66 MHz with 8MB of main memory. The

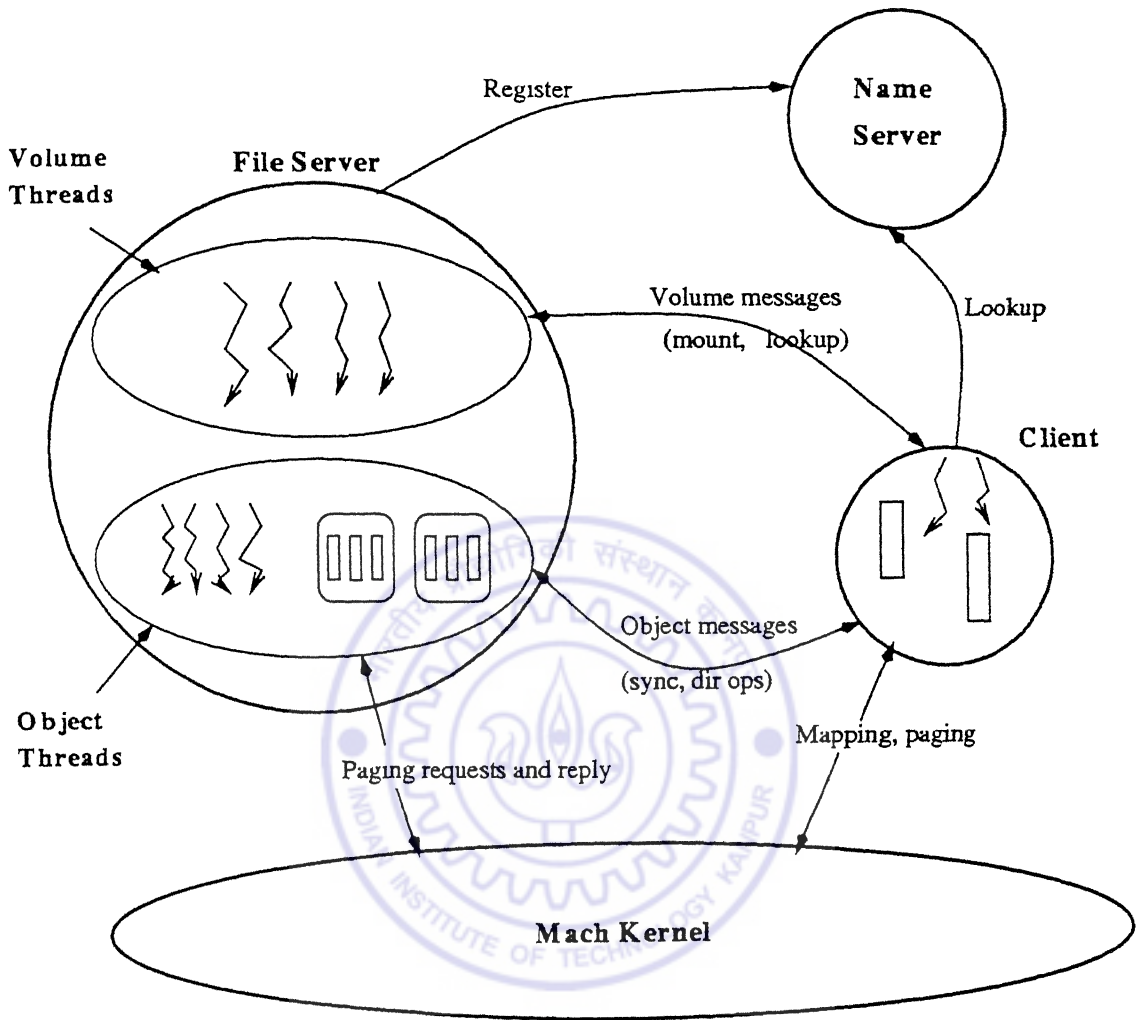


Figure 7: Mach based GFS Implementation

Unix call *times*² was used to take the benchmarks.³

Because of two user-level programs interacting in the case of Lites and GFS benchmarks we analyze only the real time, that is the wall clock time in the first two benchmarks. Though the benchmarks were taken on a lightly loaded system with only the system daemons were present in addition to the servers, the benchmarks are likely to be a bit inaccurate because of operating system overheads. However since these benchmarks are only representative they are quite sufficient to provide a good comparison.

²*times* reports user, system and real times for a process and its children.

³All timings are reported in units of *clock ticks*, which is 10ms on both the *Lites* and *Unix*.

6.5.1 *open* and *close*

The following table reports the timings for *open* and *close*. It may be noted that GFS timings include overheads due to PFS (based on the Lites file system using Lites system calls) and therefore do not represent the correct timings for GFS alone. The timings are therefore only for reference because they were subtracted from *read/write* timings in the next section. The timings are the averages of 10 trials with 1000 opens and closes in each trial. Table 1 gives the benchmark results.

Benchmark	CPU	Real
NetBSD syscall	21.70	21.60
NetBSD stdio	25.80	26.00
Lites syscall	31.60	92.70
Lites stdio	32.90	96.90
GFS User-library	71.60	238.95

Table 1: Comparison of *open* and *close* times

The difference in timings between NetBSD syscall and Lites syscall is because NetBSD is a monolithic kernel while Lites is a user-level server on Mach. *Dean* and *Armand* [DA92] gives comprehensive benchmarks on RPC call timings on Mach and Chorus microkernels. The 2.6 times increase in the cost of the GFS file server benchmark is an artifact of the *PFS* implementation. In this benchmark, with the pathname having two components, the *PFS open* does two *stat* system calls and *open* system call on the Lites Unix file system.⁴

6.5.2 Uncached *read*

It is difficult to measure the timings for an uncached *read* for small data sizes, so this benchmark reports the timings for an uncached *read* for 100 pages (409600) bytes from GFS. The cached⁵ *read* for Lites and NetBSD is also given for comparison for comparison and analysis. Write timings have not been compared because the BSD

⁴Two *stat* calls cost approximately 100 clock ticks, so the GFS *open* takes effectively 45 clock ticks.

⁵That is, present in the buffer cache.

write system call is always synchronous and the GFS *write* is always asynchronous. Table 2 gives the benchmark results.

Benchmark	CPU	Real
NetBSD syscall cached	4.00	100.88
NetBSD stdio (cached)	6.33	100.00
Lites syscall cached	14.00	69.00
Lites stdio (cached)	11.33	76.00
GFS User Library (uncached)	4.00	89.00

Table 2: Comparison of *read* times for 100 uncached pages

The real timings in this uncached benchmark varies very widely for Lites server. The GFS benchmark also varies but not as much as the Lites case. We haven't investigated the reasons for this, however the cause is most likely due to operating system overheads.⁶ The average of the lowest timings in 10 trials were taken for all the cases.

Since the PFS actually does a read system call to the Lites server, subtracting this from the GFS benchmark shows that the overhead due to a Mach RPC and *FileServer* is very less (200ms in comparison to the Lites case of 690ms to get it from the buffer cache). The NetBSD real time is surprisingly poorer than both Lites and GFS in this benchmark.

6.5.3 Cached *read/write*

The most important benchmark is the cached read, in the case of Unix or Lites from the buffer cache and in the case of the GFS from the virtual memory cache (VM cache). The performance of a raw *memcpy* is also shown for comparison. Lites has not been compared here because of two reasons, firstly we are actually measuring the page fault overhead and it is reasonable to compare it with a monolithic kernel and secondly Lites benchmark results were found to show a great deal of variation between different trials. The time reported is the average time for 100 *reads/writes* in 10 trials.

⁶The CPU timings did not show this much variation, thereby justifying this claim.

In the GFS case both *read* and *write* is a touch of the first word of each page. Touching a page is a fair comparison because once a page is touched the entire page is completely mapped into user space. Similarly modifying the first byte will cause the entire page to be written back. The results are shown in Tables 3 and 4.

Benchmark	<i>memcpy</i>		NetBSD syscall		NetBSD stdio		GFS User-Lib	
	CPU	Real	CPU	Real	CPU	Real	CPU	Real
1 byte	0.10	0.00	2.25	2.24	8.02	7.90	4.65	5.47
1 kbyte	0.27	0.27	2.83	2.79	8.17	8.22	4.67	5.48
1 page	0.93	0.98	4.60	4.57	9.89	9.88	4.68	5.62
1 page + 1	0.93	0.99	5.73	5.74	9.63	9.67	5.14	6.02
2 pages	1.94	1.99	7.63	7.60	11.76	11.81	5.42	6.24
4 pages	3.91	3.90	14.93	14.93	22.62	22.62	6.53	7.54
6 pages	5.60	6.00	22.65	22.34	38.62	38.90	7.39	9.55
8 pages	7.10	8.20	28.55	28.44	41.72	41.70	8.79	10.05
10 pages	9.20	9.80	34.75	34.34	51.72	51.70	9.49	11.25
16 pages	15.00	15.50	58.15	58.54	82.52	82.20	13.39	15.35

Table 3: Comparison of *read* times for cached pages (100 *reads* each)

The NetBSD *write* times are much more than the *read* times because *writes* are synchronous. The *read* benchmark shows that cost of a memory mapping operation is much faster reading data from the buffer-cache using a system call. It can also be seen that *stdio* overheads are significant in most cases because of the extra copy.

It is also interesting to note that for reads more than 8 pages (64 Kbytes) the GFS user-library costs less than a *memcpy*. This is because all copies have been eliminated in memory-mapping and the only overheads are the page table manipulation by the kernel at page fault time and the user-library pointer management overheads. The raw *page-in* performance is even better (See Table 5), giving better results than *memcpy* for just 4 pages. In an embedded system this is likely to improve further because context switch times due to changing protection domains (from user to kernel and back) can be minimized by moving all processes into kernel space.

Benchmark	NetBSD syscall		NetBSD stdio		GFS User-Lib	
	CPU	Real	CPU	Real	CPU	Real
1 byte	3.95	4.04	10.12	10.50	5.05	5.07
1 kbyte	5.17	5.11	11.79	11.76	5.17	5.09
1 page	8.20	8.21	16.36	16.35	5.24	5.13
1 page + 1	10.80	10.80	25.30	25.31	5.71	5.75
2 pages	19.26	164.61	34.66	164.19	5.73	5.98
4 pages	32.48	332.09	67.11	331.27	7.36	7.18
6 pages	50.85	497.94	100.82	499.20	8.14	9.50
8 pages	63.65	665.04	118.52	664.50	7.14	9.90
10 pages	77.95	873.34	205.32	784.30	9.44	11.11
16 pages	125.65	1387.24	333.32	1227.80	14.94	14.40

Table 4: Comparison of *write* times for cached pages (100 *writes* each)

6.6 User-Library Overheads

The overhead of the user-library over directly using the pointer returned by *vm_map* is shown in the Table 5 (All reported timings in Section 6.5 were using the user-library). The benchmarks show that library overheads are significant for small sizes but is relatively less for larger sizes. The overhead is because a library function call is made for every page. Since profiling support is not included in the current Mach implementation, the bottle neck could not be identified, the timings are likely to improve after proper profiling and tuning.

Benchmark	Raw <i>vm_map</i>		<i>User-Library</i>	
	CPU	Real	CPU	Real
1 byte	1.36	2.43	4.65	5.47
1 kbyte	1.35	2.34	4.67	5.48
1 page	1.77	2.40	4.68	5.62
1 page + 1	1.88	2.37	5.14	6.02
2 pages	1.92	3.03	5.42	6.24
4 pages	3.59	4.29	6.53	7.54
6 pages	4.29	5.55	7.39	9.55
8 pages	5.09	6.15	8.79	10.05
10 pages	6.79	7.35	9.49	11.25
16 pages	9.69	10.35	13.39	15.35
32 pages	17.29	18.75	22.89	24.35
64 pages	33.29	34.45	46.39	43.55
100 pages	52.39	53.35	65.19	65.65
200 pages	101.09	102.35	124.69	126.25

Table 5: Overheads due to *GFS User-Library*

Chapter 7

Conclusion

A generic file system interface for embedded kernels has been designed based on a memory mapping interface. Interfaces for both the file system implementor and the File System user have been presented in detail. An multi-threaded implementation of the interface has been done on the Mach micro-kernel, with relatively fine grain locking¹. Benchmark results on a Pseudo File System (PFS) were presented and prove that the *memory-mapping* interface is extremely efficient.

7.1 Analysis of the Design

The design has been done from both the File System implementor's perspective and the file system user's perspective. The multi-threaded design ensures that the implementation is scalable. The interface is also flexible in the sense that individual file systems can be implemented as separate servers if necessary because the file system name space has been completely partitioned.

7.2 Future Work

Because of lack of profiling tools for multi-threaded programs in the University of Utah version of the Mach kernel it was not possible to study if there were any

¹There is a lock for every object

bottle necks in *FileServer* implementation. Once this support is available profiling of *FileServer* should be done. A flexible and efficient design has been presented. Further work on GFS can be in two directions.

One is to design an efficient file system for embedded kernels using the GFS interfaces or porting an existing file system implementation to the GFS framework. The second is to improve the performance of the interface, so that overheads are further minimized. Cost of an RPC on Mach is still significant compared to a system call in a monolithic kernel, the GFS server (*FileServer*) should be moved into the kernel once support becomes available in the Mach microkernel.



Bibliography

- [ABB⁺86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *USENIX Conference Proceedings*, pages 93–112, Atlanta, GA, Summer 1986. USENIX.
- [B⁺93] Boykin et al. *Programming under Mach*. System Programmer's series. Addison-Wesley, Reading, MA, first edition, 1993.
- [Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1986.
- [DA92] Randall W. Dean and Francois Armand. Data movement in kernelized systems. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 243–262, Seattle, WA, April 27-28 1992. USENIX.
- [ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, USA, 1990.
- [GMS87] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual Memory Architecture in SunOS. In *USENIX Conference Proceedings*, pages 81–94, Phoenix, AZ, Summer 1987. USENIX.
- [Hel94] Johannes V. Helander. Unix under Mach, The LITES Server. Master's thesis, Helsinki University of Technology, Helsinki, 1994.
- [Hil92] Dan Hildebrand. An Architectural Overview of QNX. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 27-28 1992. USENIX.

- [JCS+] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. *Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Status*. OSF and CMU.
- [Kle86] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Conference Proceedings*, pages 238–247, Atlanta, GA, Summer 1986. USENIX.
- [KSU92] Orran Krieger, Michael Stumm, and Ron Unrau. Exploiting the advantages of mapped files for stream I/O. In *USENIX Conference Proceedings*, pages 27–42, San Francisco, CA, Winter 1992. USENIX.
- [KV91] David G. Korn and K. Phong Vo. SFIO: Safe/Fast String/File IO. In *USENIX Conference Proceedings*, pages 235–256, Nashville, TN, Summer 1991. USENIX.
- [Lew91] Donald A. Lewine. *POSIX programmer's guide: writing portable UNIX programs with the POSIX.1 standard*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991.
- [Loe91a] Keith Loepere. *Mach 3 Kernel Interfaces*. OSF and CMU, 1991.
- [Loe91b] Keith Loepere. *Mach 3 Kernel Principles*. OSF and CMU, 1991.
- [Loe91c] Keith Loepere. *Server Writer's Guide*. OSF and CMU, 1991.
- {Loe91d} Keith Loepere. *Server Writer's Interfaces*. OSF and CMU, 1991.
- [R+88] Richard Rashid et al. Machine independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–907, 1988. Also appeared in Proceedings of the Second ACM Symposium on ASPLOS, 1987.
- [Sta90] Richard Stallman. Towards a New Strategy in OS Design. In the GNU Bulletin, 1990.

- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, second edition, 1991.
- [Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, USA, first edition, 1994.

