An FFS-based Object Store Supporting Class Fragmentation

by

Lan Guo Scott

A thesis
Presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada

ISBN 0-612-16275-3

Name ____

Dissertation Abstracts International and *Masters Abstracts International* are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

Computer Science
SUBJECT TERM

0984
SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

AN FFS-BASED OBJECT STORE SUPPORTING CLASS FRAGMENTATION

BY

LAN GUO SCOTT

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

Persistent object systems attempt to hide the traditional distinction between short-term and long-term storage from application programmers. There are many advantages when a programmer can operate at a level of abstraction in which such distinction does not exist. A persistent object system depends upon an object store, in part, to provide persistence.

To achieve good performance, an object store must keep related objects physically close to each other in secondary storage. In an object system, class fragmentation, which is performed according to a query model of class accesses, may be used as the clustering technique to group related data together. Class fragmentation based clustering will reduce the amount of irrelevant data accessed at a local site and the amount of data transferred unnecessarily between distributed sites.

Most existing object stores are built on conventional operating systems or architectures which are inappropriate bases for persistent object systems. The object store presented in this thesis is built directly on top of the Mach microkernel in a 64-bit addressing space. The object store implementation re-uses part of the Berkeley Unix Fast File System (FFS) code. This strategy decreases the implementation complexity and takes advantage of the FFS in optimizing disk allocation. Supported by these advanced architectures, the object store will provide better performance than conventional stores.

Acknowledgments

I would like to express my deep gratitude to Dr. Peter Graham for his suggesting the topic of this research, and for his supervision, encouragement and patience throughout the course of the thesis work. The time, discussion of research methods, and comments on early versions of the thesis Dr. Graham afforded to me have been very valuable and important.

I would also like to thank Dr. Kenneth E. Barker for his precious advice at the beginning when I started my Master's program. His continuous concern and support for the progress of my research is much appreciated.

Finally, I would like to thank Dr. David C. Blight of the Faculty of Engineering for providing helpful comments and serving on my advisory committee.

Contents

List of Figures

Chapter 1

Introduction

Persistent object-oriented programming, combined with distribution technology, is increasingly being recognized as valuable for supporting large, extensible, flexible, and long-lived software. In recent years, considerable research has been devoted to the investigation of the concept of persistence [Atk83, , Mor90, Kho93, Atk94, Bil94, Ozs94] and its application to the integration of databases and programming languages, both in object-oriented systems [Mai86, And87, Deu90, Bar92, Ber92, Bro92, Ngu92, Vau92, Bil93, Cas93, Han93, Mil93, Sin93, Str93, Che94] and distributed systems [Ber92, Tri92, Vau92, Cas93, Mil93, Sou93, Che94].

A persistent object system provides an environment where objects are allowed to persist for an arbitrary length of time, possibly longer than the life time of the creating program, and where they can be accessed and manipulated in a uniform manner. There are considerable advantages when programmers can operate at a level of abstraction in which

there is no distinction between short-term and long-term data storage. In such systems, since programmers do not have to implement the explicit loading and saving of data, program development is easier. Additionally, there is no need for the explicit conversion of data from one (in-memory) format to another (on-disk) format.

A persistent object system depends on an *object store* to provide storage for objects. The development of an advanced object store is the subject of this thesis.

1.1 Motivation

An object store is used to add the features of persistence and enhanced sharing to language defined objects. It provides persistent storage (on disk or other form of non-volatile secondary memory) together with facilities for manipulating and organizing the stored objects. Existing implementations of object stores typically suffer from two inadequacies; implementation over expensive operating system bases and lack of support for exploiting object semantics. The proposed object store addresses both these issues.

Support for the efficient storage of, and access to, objects in a persistent object system is crucial to good performance in the resulting system. Conventional object-oriented programming systems which use traditional file systems do not provide adequate support for storing and sharing the objects used in application programs. There has been much work undertaken on both centralized and, to a lesser extent, distributed persistent object stores [Car90, Deu90, Kim90a, Eli90, Bro89, Ber92, Bro92, Cas93, Mil93, Mun93, Sou93, Oli94, Sub94, Yan94].

Most existing object stores are constructed on top of conventional operating systems such as Unix. Such conventional architectures provide a less than ideal base for persistent object systems because of their monolithic structure and *heavy weight* support for network communication. The recent move towards microkernel architectures has had the positive effect of improving the situation. The Mach [Tev89, Bar90, Tan92, Boy93] operating system, for example, is explicitly targeted at distributed and multiprocessor environments. Furthermore, its microkernel structure provides only the minimal required system support and does so at the smallest possible cost. Built above Mach, a distributed persistent object system can be expected to achieve better performance. For this reason, the proposed object store will be constructed on top of the bare Mach microkernel.

To obtain better performance, the object store must also keep related objects physically as close as possible to each other in secondary storage thus exploiting a form of locality of reference. To accomplish this, some form of object *clustering* is required whereby related objects are placed together in a single "cluster". One way to define the membership in such clusters is through the use of fragmentation[Kar94, Eze94a, Eze94b, Eze95]. Based on a query model of class accesses, fragmentation breaks a class (the set of all objects instantiated from a given object type) into a collection of fragments with only a subset of the original class's components. Each fragment defines a "cluster" and the parts of a fragment are co-located on disk. Similarly, fragments which are strongly related to one another may also be stored close to one another. In this way, both the time required to access objects in any given query and, because of the way fragmentation is done, also the amount of irrelevant data accessed in each query, are minimized.

Each storage unit in the proposed object store will contain a distinct class fragment, rather than an individual object. Supporting fragments allows the object store to have better overall efficiency in both centralized, and particularly, distributed environments.

1.2 Overview of the Proposed Object Store

The implemented object store will serve as the initial persistence mechanism for a large ongoing research project investigating persistent and distributed objects supported via the use of distributed shared virtual memory. The project's goal is to support a persistent, distributed object system in a single shared 64-bit distributed virtual address space. The concept of using a single shared address space is based on the work of Chase et al. [Cha92, Cha94] at the University of Washington. In addition to the initial design and implementation of the object store, extensions to support distribution are considered, although not implemented. The implementation environment for the proposed object store is the bare Mach microkernel.

File systems are one of the most commonly used data management systems. To decrease the implementation complexity and take advantage of the extensive work done previously in optimizing disk access time within file systems, the proposed object store will be built re-using as much existing code as possible.

One of the most widely used file systems, the Berkeley Unix Fast File System (FFS) [Lef89], will be adopted as the framework for storage management in the object store. Berkeley Unix which contains the modified 4.3 BSD implementation of the FFS can

be emulated above the Mach microkernel. This technique not only provides Unix compatibility on Mach, but also enhanced performance which exceeds the performance of the original 4.3 BSD implementation [Tev89]. Elements of the FFS will be used to implement the proposed object store to take advantage of these benefits and to avoid re-inventing the wheel.

A class can be fragmented vertically, horizontally, or in a hybrid way [Eze95]. Such fragmentation generates logical fragments which, it is assumed, will normally be referenced in their entirety by applications. Since for the same class, horizontal fragments and vertical fragments overlap, these logical fragments need to be further decomposed into physical fragments. Using physical fragments as storage units, there is little data redundancy in the store. Parts of the FFS are suitable for storing physical fragments.

The inode and block structures of the FFS can be applied to the storage of physical fragments. Each physical fragment will be assigned one inode number and stored as one or more associated data blocks. Thus, the implementation will re-use much of the *low-level* code from the FFS. Object specific higher level code will then be added to tailor the FFS to class fragment storage. Access to physical fragments using the FFS code in the implementation of the object store will avoid the traditional costs associated with making Unix system calls since Mach is specifically designed to efficiently support non-kernel services such as file systems.

Two levels of data access will be provided to the object store user. The first is logical fragment access which is for class-based queries which will take advantage of the

co-location of logical fragments and their contents when it is advantageous[1] to do so. The second form of access corresponds to individual object access as would be likely to occur in interactive design environments. Accordingly, the object store implementation will provide two APIs (Application Programming Interfaces), one for each form of access.

Between the storage mechanism (inodes) and the APIs, index structures are built within specific files to map logical fragments and objects to the necessary sets of physical fragments (or parts thereof). The index structures will be implemented using B^+-trees.

1.3 Model Assumptions

The design and implementation of the proposed object store is affected by certain assumptions about the underlying objects and classes. These include a set of object-oriented concepts and a set of class fragmentation concepts.

1.3.1 Object Model

In constructing the object model, the definitions of core object modeling concepts given by Kim [Kim90a] are followed. These are informally defined below:

Definition 1.1 *Objects and Object Identifiers.* Any real-world entity modeled in the system is an *object* and has an associated system-wide unique *identifier*, its "OID".

[1] This applies to class based queries.

Definition 1.2 *Attributes and Methods.* An object has one or more *attributes*, which store its current state, and one or more *methods* which operate on the values of the attributes to accomplish state transitions.

Definition 1.3 *Encapsulation and Message Passing.* Messages are sent to an object to indirectly manipulate the values of the *attributes* by invoking the methods encapsulated in the object. There is no way to access an object except through the public method interface specified for it.

Definition 1.4 *Class.* All objects which share the same attribute types and set of methods are grouped into a *class*. An object belongs to only one class as an instance of it. Each class is uniquely identified by a system-wide unique identifier, its "CID".

Definition 1.5 *Class Hierarchy and Inheritance.* The classes in a system form a *hierarchy* such that, for a class C and a set of lower-level "sub" classes $\{Si\}$ based on C, a class in the set $\{Si\}$ is said to be a specialization of the class C, and conversely the class C is said to be the generalization of the classes in the set $\{Si\}$. The classes in $\{Si\}$ are *subclasses* of the class C; and the class C is a *superclass* of the classes in $\{Si\}$. Any class in $\{Si\}$ inherits all the attributes and methods of the class C and may add additional attributes and methods or redefine existing ones. All attributes and methods defined for a class C are inherited into all its subclasses transitively. An instance of a class S is also logically an instance of all superclasses of S.

1.3.2 Fragment Model

In constructing the fragment model, based on the definitions of class fragmentation

concepts given by Ezeife and Barker [Eze95], three tighter definitions are provided below. Given a class C = (K, A, M, I) where K is the class identifier (i.e. CID), A the set of attributes, M the set of methods, and I is the set of objects instantiated using A and M, three types of fragmentation are defined on a class. A' denotes a subset of A. M' denotes a subset of M. I' denotes a subset of I.

Definition 1.6 *Horizontal Fragmentation.* A horizontal fragment, F_h = (K, A, M, I') of a class contains its class identifier, and all attributes and methods of the class but only some of its instance objects ($I' \subseteq I$) with the restriction that for any two horizontal fragments $F_{h(i)}$ and $F_{h(j)}$ of the class, $F_{h(i)} \cap F_{h(j)} = \varnothing$, where $i \neq j$.

Definition 1.7 *Vertical Fragmentation.* Each vertical fragment F^v = (K, A', M', I) of a class contains its class identifier, and all of its instance objects for only some of its methods ($M' \subseteq M$) and some of its attributes ($A' \subseteq A$) with the restriction that for any two vertical fragments $F^v_{(i)}$ and $F^v_{(j)}$ of the class, $F^v_{(i)} \cap F^v_{(j)} = \varnothing$, where $i \neq j$.

Definition 1.8 *Hybrid Fragmentation.* Each hybrid fragment F^v_h = (K, A', M', I') of a class contains its class identifier, and some of its instance objects ($I' \subseteq I$) for only some of its methods ($M' \subseteq M$), and some of its attributes ($A' \subseteq A$) with the restriction that for any two hybrid fragments $F^v_{h(i)}$ and $F^v_{h(j)}$ of the class, $F^v_{h(i)} \cap F^v_{h(j)} = \varnothing$, where $i \neq j$.

1.4 Thesis Organization

The organization of the rest of this thesis is as follows. Background material as

well as related work is discussed in Chapter 2. Chapter 3 presents an evaluation of the problems dealt with in the object store. Chapter 4 describes the object store design, while Chapter 5 addresses the implementation issues for the object store. Chapter 6 discusses considerations for extending the centralized object store into a distributed system. Finally, Chapter 7 presents conclusions, and suggests directions for future work.

Chapter 2

Background and Related Work

In this chapter, the background material used to support the persistent object system, including file systems and the Mach operating system, will be discussed. Recent research work on object stores will also be reviewed.

2.1 File Systems

To achieve persistence of the state of objects in programming languages and environments, operating system files are commonly used to store and retrieve object values. When using files, the responsibility for providing persistence rests with the object programmer who must code explicit file operations.

Files are collections of logically related information stored on a mass storage device such as a disk. The file system is the organizational framework for the collection of all such files. The following review of conventional file systems is based on [Tan92, Pet82, Lef89, Man91, Gla93].

2.1.1 General Concepts

Logical File Concepts

For users, the file system is the most visible part of an operating system. A file system is usually a collection of files and directories (repositories for collections of files) together with some operations on them. A file may be considered to be an abstract data type (ADT) defined and implemented by the operating system. From the user's viewpoint, some aspects normally attributed to files include:

- *File naming*. When a file is created, it is assigned a unique name. The file and its name still exist even after the process which created it terminates (i.e. persistence). In this way the file can be accessed by other processes through its name.
- *File structure*. There are normally three kinds of file structures: byte sequence, record sequence, and index. A byte sequence file is an unstructured sequence of bytes. A record file is a sequence of records (specific, typed collections of bytes) with certain lengths and internal structures. An indexed file has a logical

tree structure which provides direct access to records, each of which contains a key field in a fixed position.

- *File type*. Many operating systems support different types of files. This permits efficient and varied access to files for different purposes. For example, Unix [Rit74, Ker81] supports *regular files*, which contain data, *directories* which are used for maintaining the structure of the file system, *character special files* which are I/O related files, and *block special files* which are used to model disks.
- *File access*. Primarily there are two kinds of file access: sequential and random access.
- *File attributes*. In addition to file name and contents, a file often has other attributes, such as current file size, access rights for the file, and so on.
- *File operations*. The file system provides certain specific operations to allow users to create, modify, delete, and access files.

Directories, as special system files, are used to keep track of and group other files. Their primary function is to locate the starting point of files. Most operating systems support hierarchical directory systems. When the file system is organized as a directory tree, a pathname is needed to specify a certain file and each directory contains <pathname, disk location> pairs. Users can use system calls such as create, delete, opendir, rename, link, and so on to operate on directories.

Hardware Aspects of a File System

A file system provides long-term storage with files usually being stored on hard disks[1]. Figure 2.1 is a diagram of a typical disk architecture:

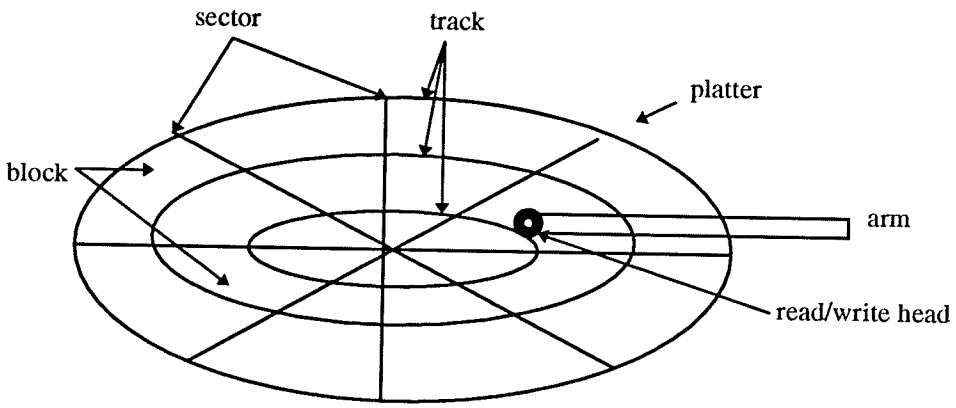


Figure 2.1 Disk Architecture

A disk is normally divided into concentric rings called tracks, and then further divided into areas called sectors. Each intersection of sectors and tracks is called a block, which is the basic unit of disk storage. A read/write head moves in and out on an arm. Information is accessed by the head when the disk rotates underneath it. The *disk controller* drives the read/write head in response to instructions from the operating system's disk device driver. Most disk drives actually contain several platters as shown in Figure 2.2. With this kind of disk, the collection of tracks in the same concentric ring is called a cylinder.

[1] Files can also be stored on other media such as tape and floppy disks, etc.

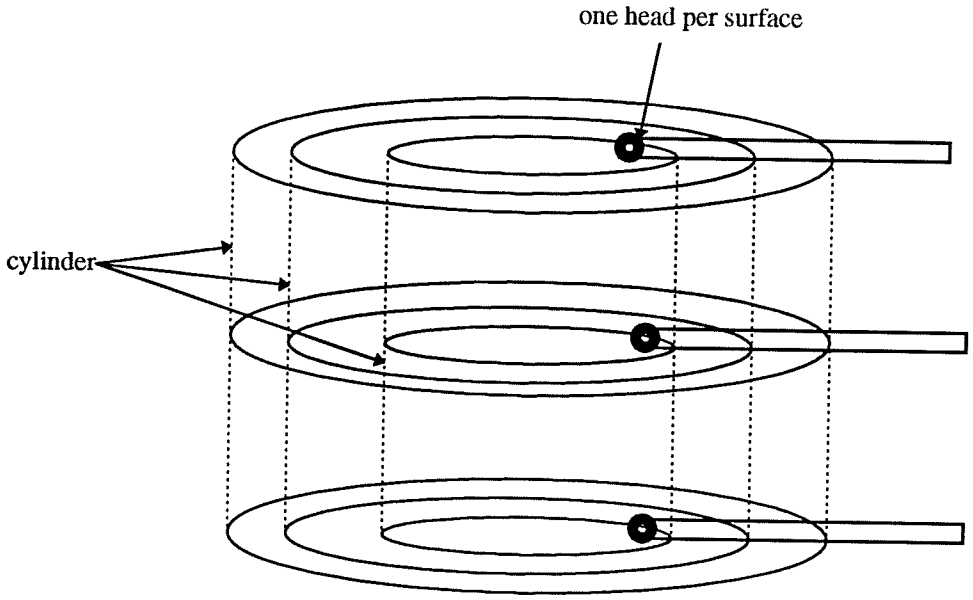


Figure 2.2 A Multi-Platter Disk

Implementation Issues

The major problem in implementing a file system is to map the logical file system structure onto the physical storage devices (disks). The physical record (block) size of devices is normally the size of a sector, but can be the size of a track, a cylinder, or the size of a page in a paging system. Block size is the unit of data transfer between memory and disk. Since the physical block size of the device and the logical record size may not be the same, most file systems block logical records into physical records. File systems must also keep track of free disk blocks. There are two common methods for a file system to

keep track of the free space in disk blocks. One is to hold the free disk blocks on a linked list. The other is to use a bit map. In the bit map method, 'n' bits are used for a n-block disk. Free blocks are represented by '1's in the map, while allocated blocks are represented by '0's.

Another important issue in implementing file storage is how to allocate the files to the free areas on disk. The simplest allocation scheme is to assign files to contiguous free blocks. This scheme is easy to implement and provides fast access. However, contiguous allocation has certain disadvantages. Specifically, the disk is increasingly fragmented due to deleting files and there is a strict requirement that file size must be known in advance. To avoid these disadvantages, many systems use non-contiguous allocation schemes, such as linked list allocation and indexes. To allow dynamic file growth and random access, Unix file systems use the i-node (index-node) method [Lef89]. Another typical file allocation structure is the B-tree. Such a system provides multi-level index structure with log search time and data stored at all tree levels. Since this scheme provides efficient manipulation algorithms and efficient utilization of space, it is widely used for representing files. OS/370 MVS supports this scheme among others [Cal82].

In a system supporting directories, the format of the directory entries must be considered. A directory entry gives the information needed to find the disk blocks of the corresponding file. Different systems have different directory entry formats. For example, in MS-DOS [Tan92] directory entries are 32 bytes long and contain the file name, attributes, and the number (i.e. address) of the first data block on disk. In the original Unix

file system, the directory entry format was 16 bytes long with the first 2 bytes for the i-node number and the remaining 14 bytes for the filename.

The key to increasing the performance of a file system is to reduce disk access time because access to disk is much slower than access to memory. One common technique is to use a block cache (also known as a buffer cache) in memory. Recently used blocks are kept in the cache. Typically, 85% of disk transfers can be avoided because the requested block is already in the cache in memory [Lef89]. To manage cache replacement, algorithms such as FIFO and LRU are often applied.

Another strategy to decrease access time is to put blocks that are likely to be accessed in sequence close to each other on disk (for example, in the same cylinder). In this way the amount of disk arm motion (i.e. seek time) can be reduced. This is important because seek time is the dominant component in disk access time.

File protection, file security, file backup and recovery are also needed and are the concern of the file system code but will not be discussed in this thesis.

2.1.2 The Berkeley Unix Fast Filesystem

Berkeley Unix is one of the most popular versions of the Unix operating system. The fast filesystem in 4.3BSD (FFS) has not only the features of traditional Unix file systems, but also many improvements. The FFS is focused on as an example of Unix file

systems because it is a component in the implementation of the object store. The following is a brief introduction of the FFS.

General Features

The Unix file system supports three kinds of files:

- Regular files (byte stream), contain either ASCII or binary data. They generally correspond to data or code.
- Directory files, are regular files with special format and interpretation that are used to group together collections of other files.
- Special files, are used to provide linkages to I/O hardware.

The Unix file system is hierarchical. Related files are grouped under directories, and the directories are organized into a hierarchical structure by nesting directories in other directories. Pathnames are used by users to access a certain file. Figure 2.3 shows an example of a nested directory structure.

Since the Unix file system is for a multiuser environment, file access restrictions are needed. A file may be accessed in one of three modes: read, write, or execute. There are also three collections of users who may accesses a file: *u*, the owner of the file; *g*, members of the same group as the file; and *o*, other users of the system. The owner/user of the file can change the access modes permitted for each of u, g, and o to control access to that file.

In Unix, file sizes may change dynamically. Users can increase the file size up to the limit induced by the amount of available storage[2]. This is one of the important features of the Unix file system.

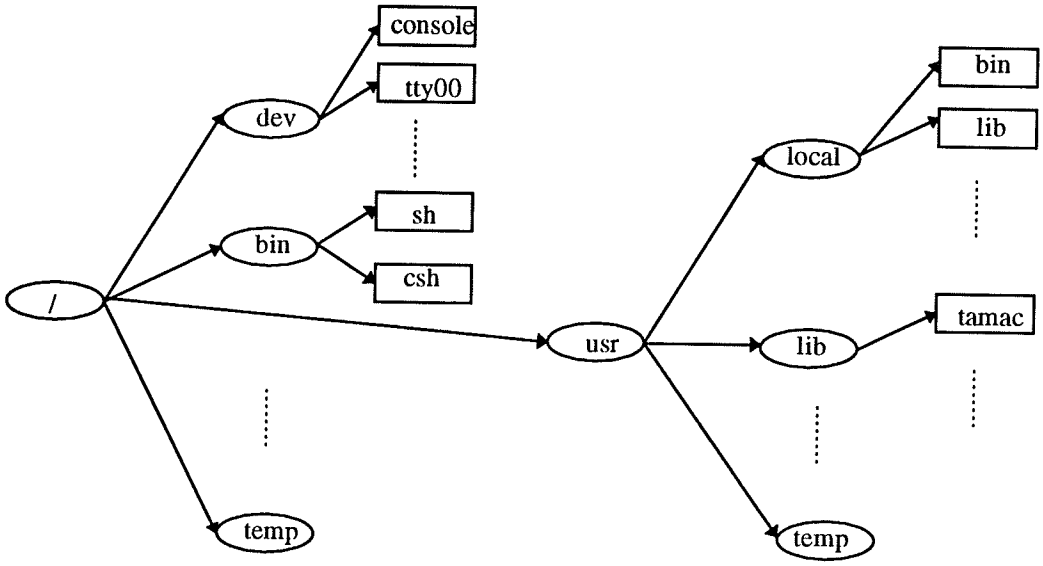


Figure 2.3 Unix Directory Structure

File Structures on Disk

4.3BSD divides each disk into one or more partitions or logical disks. Each such logical disk contains a single filesystem and in turn is divided into one or more areas called cylinder groups. Each of these cylinder groups occupies one or more consecutive cylinders

[2] File size may also be further constrained by certain implementation details.

of the disk so that disk accesses within the cylinder group require minimal disk head movement.

A cylinder group consists of an information header, and some data blocks which take up most of the cylinder group. The information header includes a boot block, a superblock, a cylinder block and an array of inodes (Figure 2.4). The boot block contains a boot strap program. The superblock which is identical for each cylinder group, consists of static parameters of the file system such as the size of the file system and the block size for the data. The cylinder block consists of dynamic parameters of the cylinder group such

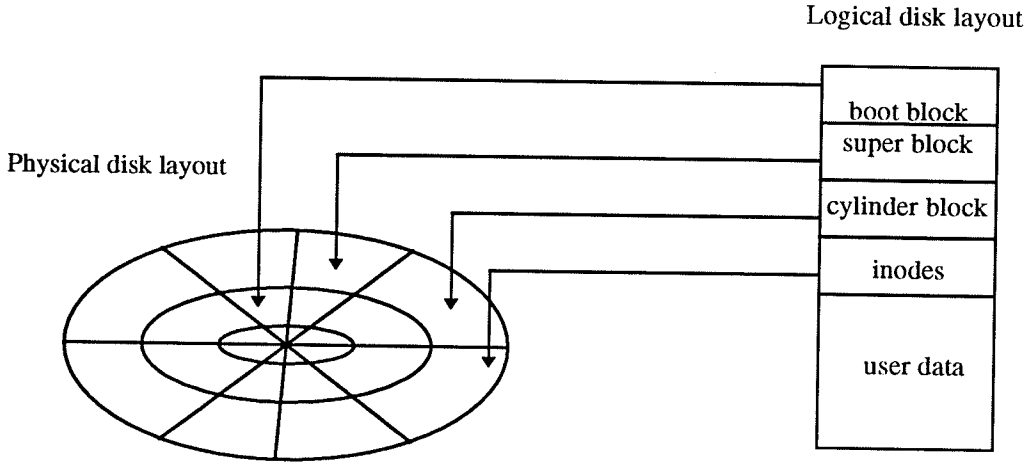


Figure 2.4 Usage of Disk Blocks

as bit maps for free blocks and for free inodes. The array of inodes stores information about each file on the disk. The Unix file system associates each file with an i-node which

keeps track of which disk blocks belong to the file. Each inode contains the following information [Lef89]:

- Logical-to physical block mapping;
- The file's owner and group-access identifiers;
- The time the file was last read and written, and the time the inode was last updated;
- The size of the file in bytes;
- The number of references to the file;
- The number of physical blocks used by the file; and
- The addresses of the file's disk blocks.

Figure 2.5 shows the structure of an inode. Each inode has 13 pointers to its file's data blocks on the disk. The first 10 pointers directly contain addresses of data blocks. The next contains the disk address of a single indirect block. Beyond that, a pointer points to indirect block, which in turn contains two indirect blocks. The last pointer is a triple indirect pointer. Since the file offset in the file structure is kept in a 32-bit word, if the block size is set to 4096 bytes, there will be no need for the triple indirect block. This is because a 2^{32}-byte file (the current maximum file size on Unix) will only use double indirection (see Figure 2.6). All the i-nodes of currently opened files are put in an in-memory kernel data structure, the i-node table to enhance performance. The i-node table is effectively an i-node cache.

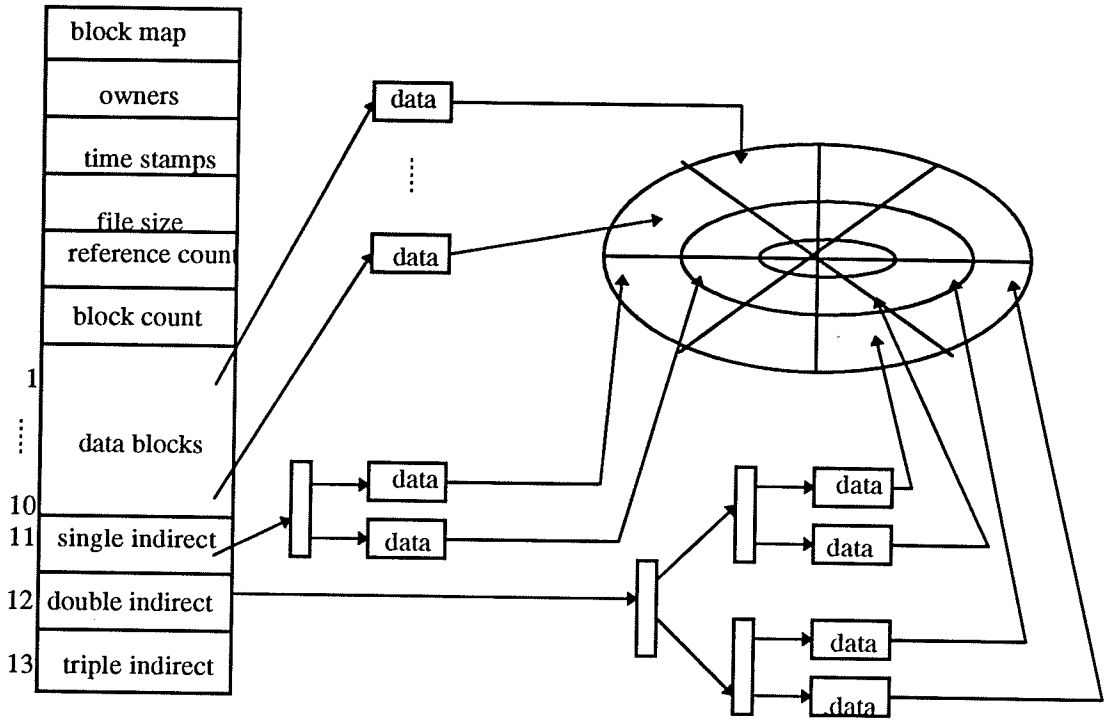


Figure 2.5 Structure of an Inode

Access Type	Bytes Accessible
direct blocks	49,152
single indirect blocks	4,294,304
double indirect blocks	4,294,967,296
total	$4,299,210,752 > 2^{32}$

Figure 2.6 Inode Data Block Capacity

In Unix file systems, directories implement a mapping between pathnames and inode numbers. When a user refers to a file, the filesystem searches for the file using the pathname supplied. Once it finds the final directory, specified in the pathname, it notes the inode number and then access that particular inode entry. Figure 2.7, based on [Gla93], shows an example of translating the pathname "top/dir2/file21" into inode number 5.

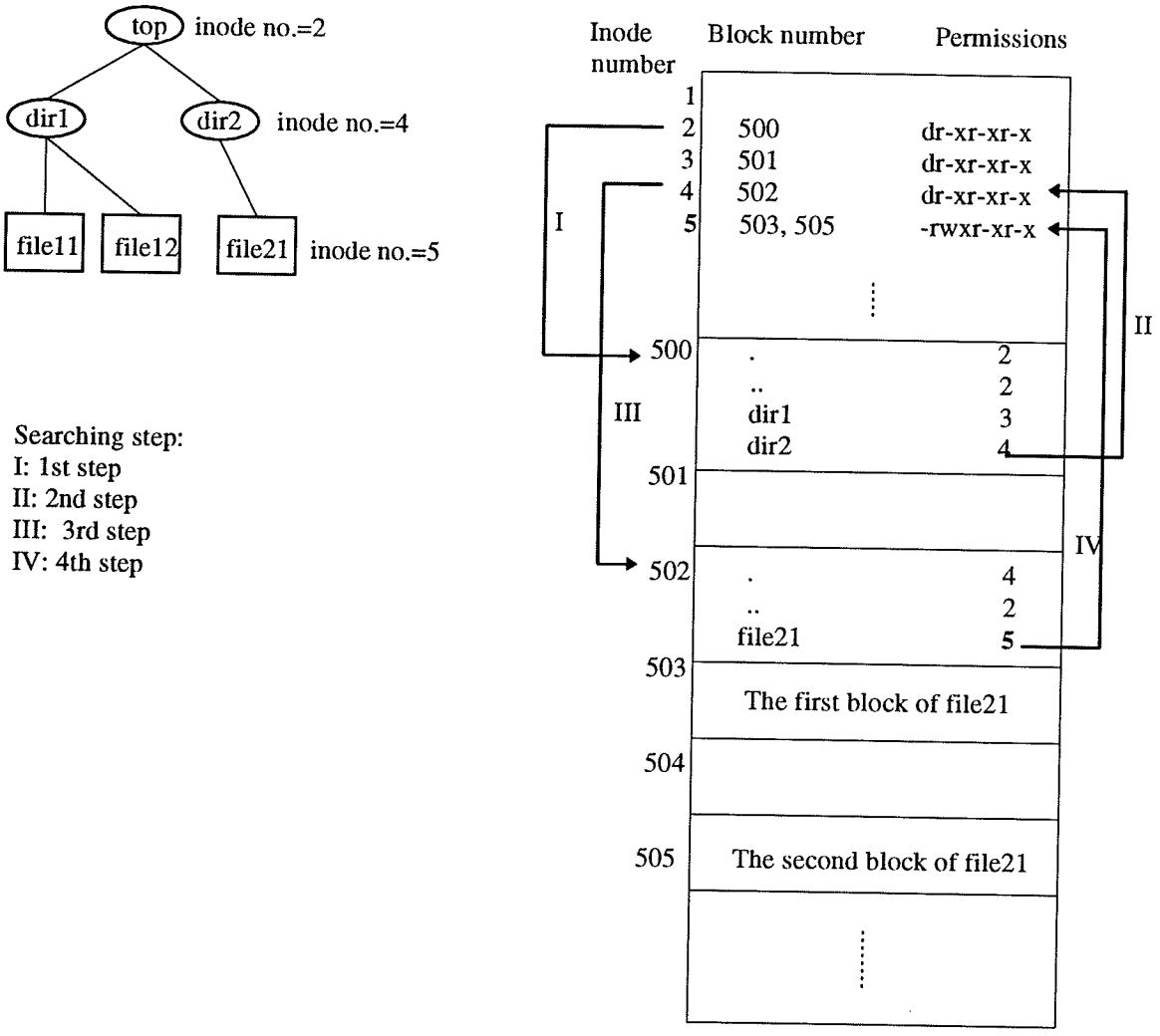


Figure 2.7 A Sample of Directory Layout

BSD Improvements over the Traditional Unix Filesystem

The Berkeley fast file system improves on the traditional Unix file system in three aspects as follows:

- The directory entry format has been extended. The limit to file name size is now 255 characters[3].
- Disks are divided into cylinder groups. The motivation for this change is to create groups of i-nodes that are close to their related data areas on the disk. Thus the i-node and the data blocks of a file are kept close to avoid long seeks.
- A two block-size strategy is introduced. For small files, using smaller size blocks reduces wasted disk space. For a larger file, using a small number of larger size blocks is more efficient than using many small blocks. Both block sizes are accommodated in 4.3 BSD.

2.2 Distributed File Systems

Since the earliest days of distributed computing, efforts have been made to allow physically distributed computers to share their data and storage resources within the same file system. This is the goal of a distributed file system (DFS). Contemporary DFSs can

[3] Each component in a pathname is a file name.

include up to a few thousand nodes (i.e. computers) on a network. Most of them, however, focus on client-server, LAN-based systems consisting of significantly fewer nodes.

2.2.1 General Concepts

In distributed file systems there are certain fundamental issues including naming and transparency, semantics of file sharing, cache related global access methods, fault tolerance and scalability [Tan92, Mul93, Lev90]. These are now briefly discussed.

Naming and Transparency

Naming is the mapping between file names and physical blocks where the file is stored. There are three common approaches to file and directory naming in a distributed system:

(1) Machine + path naming, such as */machine/path* or *machine: path* (E.g. "carbon.cs.umanitoba.ca:/usr").

(2) Mounting remote file systems onto the local file hierarchy.

(3) A single, global name space that is the same on all machines.

The first approach is clearly not location independent, since the *machine* name specifies the location. The second one is not location independent either, since it is possible to mount a given file at different places on different machines. It is not difficult to

implement the first two approaches. However, the third approach which produces a single uniform name space is a more difficult problem.

Global naming is a desirable property in a distributed system. If a user on one computer gives a name for a file, that name should have the same meaning to users on the other computers in the distributed system. This provides a measure of "transparency". Two goals in name mapping are:

- Location Transparency. The name of a file does not reveal any hint as to its physical storage location.
- Location Independency. The name of a file need not be changed when the file's physical storage location changes.

Location independency provides support for file migration (or file mobility). Location independence is a stronger property than location transparency. Naming with location independence permits the same file name to be mapped to different storage locations at different times.

Semantics of Sharing

It is necessary to define the expected semantics (or behaviour) when a file is simultaneously shared by users from different sites. Levy and Silberschatz [Lev90] identity four possible policies:

(1) Unix Semantics

- Every Read of a file sees the effects of all previous Writes performed on

that file in the DFS. In particular, Writes to an open file by a process are visible immediately by other process which have this file open at the same time regardless of their location.

- It is possible for processes to share the file pointer to the current location in the file. Thus, the advancing of the file pointer by one process affects all sharing processes.

(2) Session Semantics

- Writes to an open file are visible immediately to local processes executing on the same machine, but are invisible to remote clients who have the same file open simultaneously.
- Once a file is closed, the changes made to it are visible only in later starting "sessions". Open instances of the file do not reflect these changes.

(3) Immutable Shared File Semantics

- Once a file is declared as shared by its creator, it can no longer be modified;
- Attempts at modification result in the creation of new files. The Amoeba file system uses this scheme [Tan92].

(4) Transaction-Like Semantics

- The effects of file accesses, on a file and its output, are equivalent to the effect and output of executing the same accesses in some serial order [Nee82].

Remote Access

In a client-server system with all files stored on the server's disks, when a client requests remote access to a file, the file must first be transferred from the server's disk to the server's memory, and then from the server's memory to the client's memory via the network. If the server handles all remote access requests, unnecessary network traffic is induced. Caches can be used to improve performance by keeping the most recently used files in the client's memory. Thus when repeated access to the same data occurs, the communication overhead is reduced.

A caching scheme in a DFS must address the following design issues([Nel88]):

- *The granularity of cached data.* The granularity of the cached data can vary from small portions of a file to an entire file. If more data are cached than needed for a single access, many accesses can be served by the cached data, but unnecessary data may be transferred and stored.
- *The location of the client's cache.* File data can be cached in memory or on a local disk. Memory caches reduce access time, while disk caches increase the reliability and capacity of client machines.
- *How to propagate modifications of cached copies.* One policy is to write data through to the server's disk as soon as it is written to any cached copy. Another policy is to write the modifications to the cache and then write back to the server later. The advantage of the first policy is its reliability. The advantage of the second is that write accesses can be done more efficiently and

multiple writes to the server are not required even if the updated data is overwritten before write back.

- *How to determine if a client's cached data is consistent.* Session semantics, distributed implementation of Unix semantics, immutable shared file semantics and transactions-like semantics, help to determine cache consistency by providing constrains on when data needs to be consistent. The implementation of these semantics via consistency protocols may be expensive and non-trivial.

Fault Tolerance and Scalability

Two important benefits achievable in a distributed system are fault tolerance and scalability. A well designed DFS can tolerate faults, such as communication faults, machine failures, storage device crashes, and decay of storage media. It can also provide reliability (by replicating data) and availability (by offering access to the data through more than one computer). A well designed DFS can also expand to larger and larger systems. Such growth should have minimal expense, performance degradation, and administrative complexity. This is what is meant by scalability.

2.2.2 DFS Examples

The file system is a key component of any distributed system. With the development of distributed computing technology, more and more distributed file systems

have been developed. The following is a brief discussion of four DFSs exhibiting key characteristics of such systems [Lev90, Mul93].

The Sun's Network File System

Sun's Network File System (NFS) [Lyo85] has been widely used in industry and academia since it was introduced in 1985. Although originally based on Unix, NFS has now been ported to a wide range of non-Unix operating systems, including VMS and PC-DOS.

The goal of NFS is to allow transparent sharing among the independent file systems in a heterogeneous environment of different machines, operating systems, and network architectures. NFS does not have a global name hierarchy. Each machine has its own view of the name structure created by mounting remote file systems onto the local file hierarchy. Sharing files is on a client-server basis. NFS clients cache pages of remote files and directories in their memory. If a page is modified, it is marked as dirty and scheduled to be flushed to the server by the kernel.

NFS sites usually use a "lock manager" program to track file and record locks for consistency support when sharing files over a network. The NFS file-sharing protocol itself is designed to be stateless. That is, the servers do not hold any information about their clients. Thus the RPC requests from a client contain all the information needed to satisfy the corresponding request. Due to the statelessness, if a server crashes and then

recovers, there is no information to lose; and if the client fails, the server need not take any action.

The Sprite File System

Sprite is a distributed operating system developed at the University of California at Berkeley [Nel88, Ous88]. The goals of Sprite include efficient use of large main memories, support for multiprocessor workstations, efficient network communication and diskless operation.

The Sprite file system, including all the files and devices, appears as a single, global Unix file hierarchy. It provides distribution transparent access to files from every workstation. Unix semantics are used for sharing files.

Sprite assumes a large memory which makes it possible to use caching heavily both at servers and client machines. Sprite dynamically partitions the physical memory between the virtual memory system and the file cache and uses ordinary files as backing store for the data and stacks of running processes. This simplifies process migration since the backing files storing in the process' virtual memory are visible to all the other workstations. File system performance in Sprite is good. Normally it is faster for a client to read from a server's cache than from a local disk. This trend will likely continue as the speed of networks is increasing faster than the speed of disks.

In Sprite, each server can respond to location queries by using remote links which are pointers to files at other servers. Each client has a local prefix table which provides a

facility for mapping certain subtrees of the file system[4] to servers. Each prefix table entry contains a prefix (the topmost directory name of a file system subtree) a server (the network address of the server), and a designator (an index into the server's table of open files). Every lookup operation for an absolute pathname starts with the client searching its prefix table. The client strips the longest matching prefix from the file name and sends the remainder of the name to server specified in the prefix table along with the file system designator. The server uses this designator to locate the file system's root directory, and then uses the usual Unix pathname translation for the remainder of the file name. If a client tries to open a file and gets no response from the server, it invalidates the prefix table entry and issues a broadcast query to replace it. If the server has become available again or has been replaced, it responds to the broadcast and the prefix table entry is re-established.

The Andrew File System

The Andrew file system (AFS) [How88] is a distributed file system designed to be heterogeneous and scalable, which runs efficiently in a wide area environment on many variants of Unix.

Andrew's name space is partitioned into a local name space and a shared name space. The local name space contains the root file system of a workstation and is stored on local disks. Servers are collectively responsible for the storage and management of the shared name space. Clients can access the shared name space by querying a volume location database in a known server.

[4] The file systems discussed here are a tree structured.

The semantics of AFS are close to session semantics. When a file is opened, it is fetched from a server and put in its entirety on the local disk. All reads and writes operate on the cached copy. When the file is closed, it is uploaded back to the server. Thus, updates are visible across the network only after the file has been closed. But on a single machine, any write operation is visible immediately after it completes.

AFS is designed to work over wide-area networks with potentially many clients. Minimizing system wide knowledge and changes is important for making such a system scalable. AFS caches whole files to local disks to reduce server load. A mechanism, called *callback*, was invented to reduce the number of cache validation requests received by servers. *Callback* assumes that all the cached entries are valid unless notified otherwise. Andrew's descendent, the Coda file system [Sat90], provides high data availability while retaining the scalability of Andrew. It has stronger fault tolerance.

Locus

Differing from many existing distributed systems which have a client-server architecture, Locus [Wal83] was aimed at building a truly distributed, peer to peer operating system. The core of Locus is its distributed file system.

Locus' file naming is fully location transparent. Each file group (sub-tree of the file hierarchy) has a primary copy, the *logical* group, on a designated site and will have at least one physical file group (possibly subsets of the primary copy) on distributed site. The pair consisting of a logical group identifier and a file inode number are used as a file

identifier. Since each replica of a file has the same inode number in all physical file groups, a file identifier points to a file in general rather than a particular replica. Thus this name structure hides both location and replication details from users.

Unix semantics are used in Locus for synchronizing accesses to files. When a file is modified, the primary copy will be updated. Change messages will then be sent to all other sites where the replicas of the file reside.

Locus supports fault tolerance. When a network failure occurs, the network will be disconnected into a collection of sub-networks. As long as one copy of a file exists on a sub-network, it will be up to date with the most recent committed version, and read requests will still be served. Other copies will be updated by Locus' automatic facilities after recovery where possible.

2.3 Mach

Mach [Tev89, Bar90, Tan92, Boy93] is an advanced, microkernel-based operating system which is targeted for distributed and multiprocessor environments. As a working environment for developing application programs, Mach can be viewed as being split into two components [Bar90]:

- A small, extensible system kernel which provides process scheduling, virtual memory control, device access, and interprocess communications.

- Several, possibly parallel, operating system support environments which provide:

  (1) Distributed file access and remote execution.

  (2) Emulation for established operating system environments such as Unix.

2.3.1 Basic Kernel Functionality

The kernel functions of Mach can be divided into five categories [Bar90]:

- Basic message primitives and support facilities;
- Port and port set management facilities;
- Task and thread creation and management facilities;
- Virtual memory management functions;
- Operations on memory objects.

The fundamental abstractions which the Mach kernel supports are the following:

Task

A task is an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space (potentially sparse) and protected access to system resources (such as processors, port capabilities, and virtual memory)

Thread

A thread is the basic unit of execution. It consists of all processor state (e.g.

hardware registers) necessary for independent execution. A thread executes in the virtual memory and port rights context of a single task. The conventional notion of a process is, in Mach, represented by a task with a single thread of control.

Port

A port is a uni-directional communication channel -- implemented as a message queue managed and protected by the kernel. A port is also the basic object reference mechanism in Mach. Ports are used to refer to objects; operations on objects are requested by sending messages to the ports which represent them.

Port set

A port set is a group of ports, implemented as a queue combining the message queues of the constituent ports. A thread may use a port set to receive a message sent to any of several ports.

Message

A message is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.

Memory object[5]

A memory object is a secondary storage object that is mapped into a task's virtual memory. Memory objects are commonly files managed by a file server, but as far

[5] The Mach object should be distinguished from an object in the persistent object store. The former is an abstraction of Mach, which can be a thread, a task, a port, or memory pages (or files). The latter is a unit of data in the object base.

as the Mach kernel is concerned, a memory object may be implemented by any object (i.e. port) that can handle requests to read and write data.

2.3.2 Memory managers

Mach provides unique memory management services for users to control virtual memory paging operations (e.g. "page in" and "page out"). These services provided, outside the kernel, are called memory managers, or external pagers, or just *pagers*. When a task requires data residing in a region of virtual memory not currently in the physical memory, a page fault will occur. At this point, the pager maps the required memory object from a disk to the task's address space. When a page's contents have been modified since its last page in, the pager is also responsible for writing the dirty page back to the disk[6].

To map a memory object to a task's address space, a pager uses ports to pass the message to the task. Three kinds of ports are needed [Tan92]:

- *Object port,* is created by the pager and will later be used by the kernel to inform the pager about the page faults and other events relating to the memory object.
- *Control port,* is created by the kernel so the pager can respond to the events.
- *Name port,* is used as a name to identify the memory object.

[6] This user level control over virtual memory operations is fundamental to the effective implementation of Distributed Shared Virtual Memory.

To perform the memory object mapping, a strict protocol must be used for communication between the kernel and the pager. This is implemented by specific system calls. For example, the system call *memory_object_data_request* returns to the kernel a specific page in response to a page fault; and the system call *memory_object_data_write* takes a page from memory and writes it out to disk.

External memory management allows Mach to be the base for implementing a page-based distributed shared memory system. Users of different machines at different sites can then view a single, linear, virtual address space. The shared memory manager, implemented as an external pager, controls which pages of memory can be accessed by which machines at specific times. The control of the shared memory manager also must enforce the consistency and security of the shared memory.

2.4 Persistent Object Systems

In this section a brief review of research work undertaken in persistent object systems in recent years is given. This presentation is divided into two parts; the first discussing object-oriented databases and the second discussing other persistent object systems focusing on those using virtual memory techniques for implementation.

2.4.1 Object-Oriented Databases

Object oriented databases are one form of persistent object system. Objects are

stored as data and manipulated by object-oriented languages. The objectbase deals with object identity, storage management, concurrency control and transaction processing [Mil93]. Many object oriented database systems exist as either commercial products or research prototypes. These include ORION [Kim90a], GemStone [Bre89], VBASE [And87], Statice [Wei88], IRIS [Wil90], O2[Deu90], Starburst[Haa90], Cactis[Hus89], ODE [Agr89], ObjectStore [Kho93], EXODUS [Car90], Mneme [Eli90], POSTGRES [Row87], and ENCORE/ObServer [Kim90b].

ORION [Kim90a, Kim90c] is a distributed object-oriented database with a simple client-server architecture. It consists of several major subsystems. The storage subsystem allocates and deallocates pages on disk, moves pages to and from disk, finds and places objects in buffers, and manages indexes on the attributes of a class. Most of the parts of the storage subsystem reside in the server. Other subsystems reside in the clients to evaluate queries and to access objects in the local object buffer pool. ORION associates each object with a globally unique identifier, UID. It consists of a pair, <class identifier, instance identifier> (in the distributed version of ORION a site identifier is also included). The storage format for object instances contains several parts:

- UID as described.
- Object length: records the total length of the object.
- Attribute count: records the number of attributes stored in the disk format.

- Attribute vector: consists of the identifiers of all attributes for which the object has explicitly specified values.
- Value-offset vector: consists of the offsets of the values of the attributes.

A class contains two types of information. One is the specification of the attributes and methods shared by all instances of the class. The other is the specification of the attributes and methods which apply to the class itself.

In a fashion similar to cylinder groups in the BSD4.3 file system, an ORION disk is divided into segment groups. Each segment consists of a few blocks or pages. To improve system performance, instances of the same class are clustered in the same segment or group of segments, and instances that belong to a user specified collection of classes are similarly stored together. The ORION storage manager is responsible for allocating, deallocating, and tracking the objects in their on-disk format. The storage manager employs an object directory to record the physical addresses of objects. ORION uses extendible hashing to maintain this object directory for quickly mapping the object identifiers to their on-disk addresses. A B^+-tree is used to maintain the class hierarchy to speed up the associated searches for objects.

In the O2 system [Deu90], persistence is implemented by associating with each object a reference count which records the number of other objects pointing at it. As long as the count is greater than 0, the object and all its components are persistent. When an

object's reference count drops to 0, this means it is unreachable from other objects. Thus, this object no longer need be stored, and the system then automatically deletes it.

The O2 storage subsystem is built on top of WiSS (the Wisconsin Storage System) [Hur93]. WiSS allows data pages to contain records of different types. It also allows a new record to be inserted in a specified location. With these two features, all records belonging to the same complex object can be clustered on the disk. O2 implements object identifiers as persistent identifiers. This is done by storing an object in a WiSS record so that the object identifier is the record identifier, RID.

O2 supports tuple, list and set structured complex objects. A tuple is represented as a record stored on a page on disk, lists are represented as ordered trees, and a set structured object is itself an object containing the object identifiers of its members. B-tree indexes are used to represent large sets. Small sets are kept ordered.

EXODUS [Car90, Eri93] is an extensible database system. Its architecture includes storage and transaction management, and the persistent programming language E (an extension of C++). The storage component of the EXODUS project is the EXODUS storage manager (SM), which provides facilities for reliably storing objects. SM has a client-server architecture. The client interface, supported by a client library, allows programs (applications) to create, destroy, modify database files containing objects and to iterate through the contents of these files. Files are implemented using B^+-tree indexes with the object page ID of the buffer pool as a key. Related objects can be placed in a common file and scanned in sequence. File operations that require accessing or changing

the B^+-tree occur on the server. Each object is referred to by an object identifier (OID). The OID of a small object directly points to the object on the disk. The OID of a complex object points to the root of the relevant B^+-tree.

An Exodus client connects to a server by using TCP sockets and RPC. Though a client can only connect to one server at a time, the SM server simultaneously supports multiple clients which may be on the same machine as the server or on different ones. The SM server supports transactions with full concurrency control and recovery[7].

The Mneme project [Eli90], which is similar to the EXODUS project, is aimed at combining a persistent programming language with database features. The Mneme persistent object store is a fundamental component of the project. The architecture of the Mneme store consists of:

- A Client code module, which includes the application program, user supplied policy routines, and optionally, a language run-time system.
- The Mneme code module, which includes a Mneme client library and default policy routines.
- The Mneme server module, which includes remote server interface, local server interface, and local operating system interface.

The connection between (1) and (2) is via the client interface, whereas the connection between (2) and (3) is via the client-server interface.

[7] Cross-server transactions are not supported.

The format of a Mneme object includes four components. First, the object identifier, OID which is a logical descriptor. It is not directly linked to a precise physical location. Second, an object may contain some OIDs to describe its aggregation relationships to other objects. These OIDs can be easily enumerated for supporting garbage collection. Third, each object has a few associated attribute bits used for indicating such properties as whether the object is read-only. Last, each object has a current size.

Mneme uses object handles to access objects. A handle includes the object's OID, a pointer to the data part of the object, and the size of the object. A handle provides efficient access to the internals of a memory resident object. Creating a handle requires the object to be located. If the object is not resident, it must be fetched from external storage. This process is called an object fault. Mneme files are the modularity units for grouping and naming objects. Within a file, a further structure, a logical collection of objects called a pool, is used. A file may contain different pools. A pool determines the policy under which the objects in it are managed.

Mneme transactions provide the concurrency control and synchronization important in a system allowing concurrent work. The client interface provides various operations, such as object handle and pointer operations, pool and strategy operations, file operations and other miscellaneous operations. The server interface is based on physical segments which group multiple objects into a single chunk for transfer and storage.

2.4.2 Persistent Virtual Memory and Object Stores

Most persistent object systems have been developed on top of conventional architectures and operating systems. To achieve acceptable performance, advanced operating system support for building persistent object stores is also being studied. Operating systems with distributed object-oriented features include Choices [Mad91], Clouds [Das92], Cool [Hab90], Soul [Sha91], Monads [Ros92] and Guide [Bal91]. Some of these support persistent object spaces [Das92, Bal91].

In systems supporting persistent object spaces, the object concept is implemented at the operating system level and the operating system presents users with a single level view of storage. For example, the operating system Choices which is written in an object-oriented language, supports distributed parallel applications on a network of multiprocessors [Cam91]. The kernel is implemented as a dynamic collection of objects that have been instantiated from system defined classes. Choices has a paged virtual memory organized around memory objects. Each memory object can have its own separate backing store, page placement and page replacement algorithms. It can be shared, both within a shared memory multiprocessor and between networked computers using a distributed virtual memory protocol. Choices supports an object-oriented file system model in which files may be mapped into virtual memory. The kernel and an object file system together provide a persistent object store.

In the recent years, many systems have been generated which support large virtual memory based on advanced architectures. Rosenberg et al. [Ros92] discuss mechanisms for supporting large virtual memories. The authors believe that a persistent object store can be implemented via an extended virtual memory with address space large enough to address all objects. The address space in such a system is very sparsely populated and the paper considers the problem of mapping large sparsely populated virtual addresses to a small, densely populated memory from both the hardware and software viewpoints. A prototype implementation is made on the experimental MONADS-PC computer system.

Much research has also been done concerning the construction of a persistent object store on top of the Mach microkernel. Chevalier et al. [Che92] describe the design and implementation of a fault tolerant storage system (Goofy) for distributed object oriented applications. Goofy supplies object storage using Guide which runs on top of the Mach 3.0 micro-kernel.

Vaughan and Smith [Vau92] describe the Casper system which uses memory mapping and shadow paging to provide a distributed resilient persistent store. Casper exploits a number of the facilities provided by Mach.

Castro et al. [Cas93] describe the architecture and implementation of MIKE, a version of the IK [Rob91] distributed persistent object-oriented programming platform built on top of the Mach microkernel. MIKE supports the abstraction of a persistent single level object store. Objects are transparently loaded on demand when first invoked and

saved to disk when the application terminates. Compared with the Unix versions of IK, MIKE achieves good performance through the use of Mach abstractions.

Millard, *et al.* [Mil93] present the design and implementation of SPOMS (Shared Persistent Object Management System) which is a memory-mapped store built on top of the Mach operating system. The authors believe that making use of operating system support for memory mapping makes the storage and manipulation of persistent objects simpler. This technique also provides sharing at a much finer granularity than what is provided by conventional database systems.

SPOMS is a run-time system that provides a store for persistent objects. Objects are created via calls to SPOMS. When objects are used, SPOMS maps them into the address spaces of the requesting processes. The Mach approach to virtual memory management permits local, single-copy sharing of code and data, object faulting and transparent, on-demand object access. The Mach external pager interface allows user-level programs called external pagers to manage objects that can be mapped into the virtual memory of a task. Once the object is mapped, page faults on this object are sent by the kernel to the port which identifies this object, and are then received by the external pager. If threads in two tasks map the same memory object, the kernel will send page fault requests for each page only once in order to maintain the consistency of the pages. Thus, a Mach-based implementation of distributed shared memory is used to provide a distributed implementation of the object store. To provide distribution, the external pagers provide

multiple-reader, single-writer coherency between memory-objects on a network of computers.

Chapter 3

Problem Assessment

Persistent object-oriented programming systems are relatively new and it is still not clear which of the many models best suits the persistence paradigm. This chapter will discuss the general issues concerning persistent object stores and examine the requirements of an object store supporting class fragmentation.

3.1 Persistence

Persistence, as an attribute, can be defined as the length of time which data exists and is usable. It is concerned with supporting the uniform treatment of data independent of its lifetime [Atk83, Mun93]. For the purpose of this dissertation, the following definition applies:

Definition 3.1 *A Persistent Object System* is a programming environment that provides objects[1] which may outlive their creating processes.

3.1.1 Degrees of Persistence

The data created and manipulated by object-oriented programming may have different degrees of persistence and can be categorized as follows [Mor90]:

- Data persistent within expression evaluation.
- Data persistent as local variables in a procedure activation.
- Data persistent as global variables and heap items whose extent is different from their scope.
- Data persistent between executions of a program.
- Data persistent between various versions of a program.
- Data persistent over the program that created it (data that outlive the program).

The persistence referred to in the first three categories, referred to as short term data, can usually be supported by any programming language. The persistence referred to in the other three categories, referred to as long term data, needs a file system or a database to support it. Programming in traditional programming languages, the programmer must explicitly include code to move long term data between memory and a file system or a database, whereas persistent programming languages can transparently provide persistence

[1] Objects and their use are described well by Booch [Boo91].

for both short term and long term data. Thus, the programmer is freed from the need to manage data storage. This leads to simpler, less error-prone programs.

3.1.2 The Advantages of Persistence

The advantages of persistence can be summarized as follows [Mun93]:

- Persistence improves programming productivity by offering simpler semantics - There is no need for the programmer to deal with long term data storage or the representation transformations that often go with it (e.g. flattening structures for on disk storage).
- Persistence removes ad hoc arrangements for data translation and long term data storage - Uniformity is provided since policies are enforced by the system and not left to individual programmers.
- Persistence increases protection over the whole environment - This also enhances uniformity of storage in the system.

The first two advantages result from eliminating the distinction between long and short term data. Databases are effectively incorporated into the programming language. It has been estimated that 30% of the code in an application is related to data movement between main and secondary memory [Atk83]. If the use of long and short term data is integrated, not only can the size of the application code be reduced, but also the time to execute the code and store the data can be reduced. Therefore significant cost saving can be achieved[Bro89, Mor90]. The third advantage occurs because all the protection

facilities of a programming language can be applied to long term data. Using a single enforced type system for all data yields enhanced protection throughout the whole system.

3.1.3 Persistent Object Identities

The fundamental goal of persistent object-oriented programming is to make persistence *orthogonal* to all objects. That is, any type of object should be allowed to be persistent. Any class should be able to instantiate an object with persistence as its property. One of the most important things for implementing a persistent object system is to implement *persistent object identities* in the system.

Definition 3.2 Each object in a persistent object system has a system-wide, unique, and persistent *Object Identity* which identifies it and distinguishes it from all other objects (Object identities may be used as "object references"/"object pointers") .

Different strategies can be applied to implementing object identity [Kho93]. Two possibilities for uniquely identifying objects are:

- An object's address such as a virtual memory address or secondary storage address may be used.
- A key identifier in an object table may be used - this can be a memory-resident object table or a disk-resident object table (see Figure 3.1).

The first scheme is probably the simplest implementation of object identity. The second scheme requires a system maintained table where each object identifier is an index into the

table or a key value which is used for lookup in the table. Each entry of the table directly or indirectly contains the object's address in virtual memory or on disk.

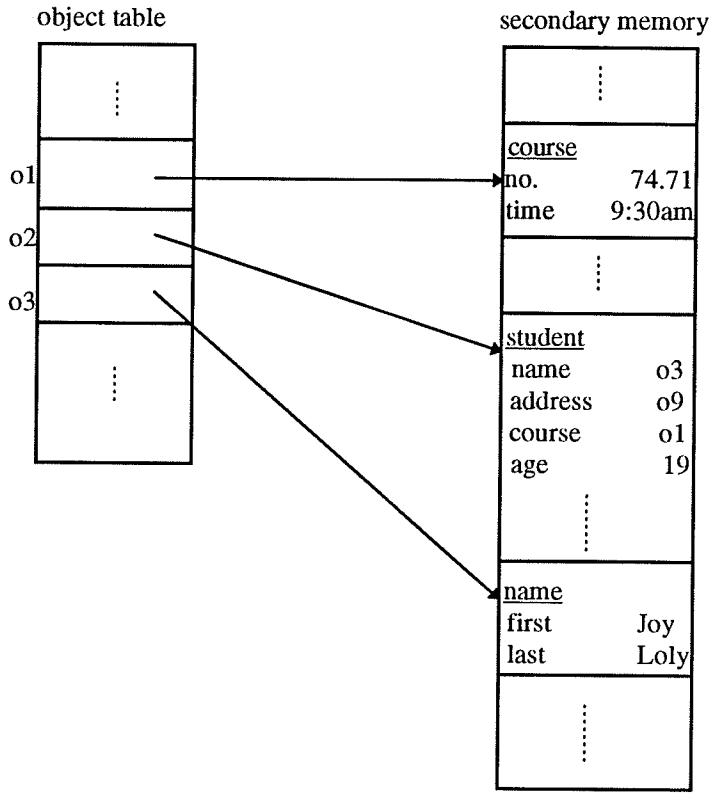


Figure 3.1 Object Identifiers as Indexes

Compared to the first scheme, this implementation of object identity has extra processing overhead, but is more flexible. When an object is moved from one position to another, either on disk or in virtual memory, its object identifier can stay the same. Only the object table is updated. The ability to freely move objects without affecting their identity is very important in garbage collection since garbage collection relies on object

movement. To collect the storage which is not in use, the usable objects need to be moved together. The second scheme is also machine independent and therefore makes it easier to port the implementation between different systems.

In the implementation of object identity in a DSVM system, persistent virtual addresses are used as object identifiers. Since the size of a virtual address is the same size as addresses generated by hardware, the need to swizzle [Kem93, Wil91] object identifiers to virtual memory addresses is eliminated.

3.2 Object Storage

Definition 3.3 Persistence of object-oriented programming relies on the existence of a repository for storing all the objects. Such a repository is referred to as an *Object Store*.

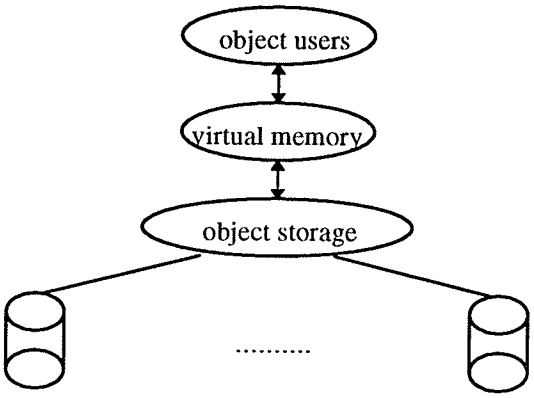


Figure 3.2 Architecture of a Persistent Object System

Figure 3.2 shows the general architecture of a persistent object system built on an object store. The *object storage* handles the object storage management of the persistent object system. Based on the *object storage,* the *virtual memory* provides users a single level view of persistent objects. On the top, the *object users* is an user interface of the persistent object system. The ideal properties of a persistent object store include:

- Infinite speed.
- Unbounded capacity.
- Total reliability.

In terms of current technology, none of these properties is realistically achievable. In the design of an object store the above attributes are goals, but in the implementation only best approximations may be provided.

3.2.1 Object Store Speed

Access speed is a crucial factor in building a successful object store. Three major strategies can be used to enhance the speed of an object store. They are:

- Constructing the store as close to the hardware as possible to minimize the inefficiency induced by the operating system;
- Supporting class fragmentation which provides efficient data manipulation and access in the store, especially for a distributed system; and

- Applying optimal data allocation polices which keep the related data physically as close as possible so that disk access times can be minimized.

3.2.2 Object Store Capacity

An ideal object store should support objects of virtually unlimited size and number. If the capacity of the store is not big enough, newly created objects can not be accommodated. Restrictions on object size limit orthogonal persistence. An object store must support reasonably large objects and must attempt to maximize the number of objects which may be stored in whatever space is available. This may be accomplished using:

- Hardware - The store may employ as many large size disks as possible to physically increase the storage capacity. In the DSVM project, 64-bit processors are used which can address 2^{64} bytes in virtual memory. With enough disk support, the system will provide very large capacity.
- Software - The object store should keep collecting free space. This is typically done through garbage collection in a dynamic system. As well, the object store must eliminate all unnecessary data redundancy, and keep needed "housekeeping" overhead to a minimum.

3.2.3 Object Store Reliability

Reliability of an object store plays an important role in supporting persistence.

System failures can cause data destruction. A persistent store should recover from any failures within the store because losing data, of course, means persistence ends. The potential failures that may occur can be categorized as follows:

- Hardware failures - occur from the breakdown of a hardware component, for instance, a disk head crash or corruption of the recording medium.
- Software failures - occur when memory is lost or corrupted.

Hardware failures are usually handled by duplicating the store after any data update. This can be done by either maintaining more than one active data copy, each of which is updated in parallel, or by maintaining a continuous log of all changes to the system [Bro89]. This issue is left as future work. To avoid complexity, it will initially be assumed that the disks are physically reliable so there are no hardware failures. The design and implementation of the object store (discussed in subsequent chapters), however, will consider how to restore the system under the circumstance of software failures.

3.3 Fragmentation

Supporting class fragmentation (see the fragment model presented in Section 1.2.2) distinguishes our object store from other existing object stores. This section presents the motivations for using this strategy.

3.3.1 Object Clustering and Class Fragmentation

Definition 3.4 *Object Clustering* is a way to group objects in secondary storage according

to common properties[2] to speed up processing on those objects.

Clustering is a very important factor in enhancing the performance of database management systems [Ber94]. Because of the *aggregation* (regarding complex objects) and *generalization* (regarding class inheritance) features of the object-oriented data model, object stores and object databases have special properties which can be exploited as the basis for clustering.

The main idea behind clustering is to group data items which are frequently accessed together so they are stored as close as possible to one another on secondary storage and thus can be retrieved quickly. Clustering techniques are important for both query-oriented object access and object navigation. Query-oriented object access refers to accessing a group of related objects. For example, two objects which are related by inheritance may be accessed together due to polymorphism. Object navigation refers to accessing a single object by navigating through object references. Object clustering enhances performance only if objects are clustered in terms of the access patterns a user will make. According to March [Mar83] and Bertino [Ber94], the task of clustering is to physically arrange the database so that:

- Obtaining the next piece of information needed by a user query has a low probability of requiring additional access to secondary storage; and
- A minimal amount of irrelevant data is transferred when secondary storage is accessed.

[2] The properties may include being subsets of objects with the same attribute types or being certain groups of objects within the same class.

For object-oriented data models, the clustering of a class may apply both to the set of instance variables and the set of methods that belong to the class. Based on the query frequency and the access locality of user applications, fragmentation breaks a class into a set of fragments with only a subset of the class' components. As presented in Chapter 1, there are three types of class fragmentation: vertical fragmentation, horizontal fragmentation and hybrid fragmentation. Fragmentation can be used as a form of clustering for object systems. When a class is instantiated with objects, the class fragments will yield the object clusters. We may regard these class fragments as a kind of object clustering between the objects within a class. Thus, each type of fragmentation defines a clustering type. Fragmentation provides a clustering scheme by fragmenting a set of objects into a set of class-fragments so as to minimize the amount of irrelevant data accessed by applications.

3.3.2 The Benefits of Fragmentation

An object store greatly benefits from fragmentation in a distributed object environment. The main advantages of fragmentation are the following [Eze95, Ozu91]:

- Using fragments reduces the amount of irrelevant data accessed by applications because different applications access or update only partial classes which can be arranged as fragments of the class.
- Fragmentation increases the level of concurrency. Decomposition of a class into fragments, each being treated as a unit, allows some data operations to

execute concurrently. In addition, if the data operations can be divided into a set of sub-operations, with each operating on a fragment, then each single operation can be executed in parallel.

- Fragmentation reduces the amount of data transferred between different network sites because each node only fetches the needed portion of the class, instead of the whole class.
- Replicating fragments is more efficient than replicating the entire class because it reduces consistency problems during updates and saves storage.

Fragmentation provides a good basis for reflecting major object-oriented features such as class inheritance and encapsulation in a logical fashion in the store. Using fragments as a basis for object clustering in an object store enables the store to support object-oriented programming more easily and naturally. Furthermore, since complex objects can be properly fragmented by the fragmentation technique, the data allocation in the object store becomes easier.

3.4 Implementation Approaches

There are three possible alternatives for implementing persistent object stores. An object store can be built on:

- A novel system designed and built from the ground up.
- An existing object database and its object model.

- An existing file system.

The first approach is the most aggressive one, which develops an entirely new object store without relying on any existing code except for the needed device drivers. Such an implementation completely meets the requirements of an object store. This "ideal" implementation offers the best possible performance since all aspects of the system can be tailored specifically to object storage. However, constructing such an object store is a major effort. It is not feasible to implement such a system in a short time.

The approach of adopting an existing object database, such as the Exodus storage manager, and modifying it to support fragments would require less work. There are, however, three reasons for not taking this approach. First, existing object databases were not designed with class fragmentation support in mind and thus it would likely be difficult to add them without making significant changes to the existing code. Second, such systems are typically built on top of Unix and thus require the support of Unix for their operation. The DSVM project runs without Unix and thus cannot provide the needed support. Finally, using existing object databases incurs the overhead related to their reliance on an existing monolithic operating system.

The third approach is to adapt a file system to manage object data. The most commonly used strategy to store and retrieve data is to use operating system files. Files are convenient units for storage, and they are reasonable units of backup, recovery, garbage collection, and transfer between different stores. The file concept is implemented by existing file systems, which offer functionality including symbolic and internal identification for ease of reference, physical location and physical organization, access

control, and various administrative operations including file creation, modification, deletion, backup and so on. Much of this functionality is also required in an object store. In addition, specific features of the object store can be added to the file system if necessary. For example, the index structures sometimes used to support object identity can be constructed. It is practical to develop the object store by utilizing a file system together with additional index structures as the data management system of the object store. Employing an existing file system which has been well tested and is reliable will significantly shorten the implementation time of the object store. Based on the advantages mentioned above, the file system approach will be applied to the construction of the object store.

Chapter 4

The Object Store Design

In this chapter, a design for an object store with integrated support for fragments is described. Aiming at achieving fast data access and optimal use of storage, this design focuses on fragment storage format, allocation policies, and data retrieval structures. A key feature of the design is to use existing system resources and facilities to implement this store on top of the Mach microkernel, instead of building everything from scratch.

4.1 Goals and Limits of the Design

The overall goal of the design is a working prototype of a fragment-based, persistent object store with support for major object-oriented features through the use of class fragmentation. Abstractly, the store will maintain two kinds of data: fragments and objects. The specific objectives of the store are as follows:

- The store should provide an appropriate notion of fragments and objects including the representation of fragments and objects, and the relationships among them.
- The store should have reasonable performance for storage and retrieval, and low overhead for data manipulation.
- The store should make the best use of available storage by reducing data redundancy to a minimum so as to maintain a large storage capacity.
- The store should offer good reliability and ensure quick recovery from system failures.
- The store should be naturally extendible to operate in a distributed environment.
- The store should be a good basis for further research (i.e. it should be flexible to support future research in integrating persistent object-oriented programming and database features).

Since this object store is an initial prototype, effort is focused on providing only the fundamentals and areas such as distribution, multiple user support, and high level database features are left to future work. The feasibility of an object store supporting fragments and providing good performance for both object and raw fragment retrieval is the chief concern. At this stage, multiuser and distributed system support are not needed. If provided, they would increase overhead and make comparison with existing centralized object stores difficult. Similarly, including high level database features such as transactions and a query interface would be inappropriate. Finally, since previous work on class

fragments assumes statically generated fragments, this object store is created statically. Making these simplifications keeps the design and implementation conceptually clear. This store will lay a reliable foundation for further development.

4.2 Physical Fragmentation

The most commonly needed class fragments (see the data model of the object store described in Chapter 1) by user applications are vertical fragments and horizontal fragments. These fragments are normally referenced in their entirety by applications. Such fragments are referred to as *logical* fragments to distinguish them from their storage forms as discussed below. Since, for the same class, vertical fragments and horizontal fragments are overlapped, storing both kinds of fragments will result in data redundancy and hence wasted storage. To avoid this, logical fragments need to be further decomposed into what will be referred to as *physical* fragments. The definition of a physical fragment is as follows:

Definition 4.1 Given the sequence of the bytes composing a horizontal fragment F_h and the sequence of the bytes composing a vertical fragment F_v of a class C, the intersection subset of F_h and F_v

$$Pf_{hv} = F_h \cap F_v$$

is a sequence of bytes referred to as a *Physical Fragment* of class C.

Each physical fragment is a subset of a vertical fragment and a horizontal fragment at the same time. When considering vertical fragments alone, each physical fragment is a subset of a vertical fragment with the same vertical fragmentation scheme of the attributes and the methods but which only groups partial object segments that the vertical fragment refers to. When considering horizontal fragments alone, each physical fragment is a subset of a horizontal fragment with the same horizontal fragmentation scheme of objects but which only groups partial attributes and methods that the horizontal fragment possesses. Therefore, if only vertical (or horizontal) fragments exist in the system, each physical fragment is the same as each vertical (or horizontal) fragment. The following is an example which illustrates physical fragmentation:

Given a class C:

{ attributes: A_1, A_2, A_3, A_4;
methods: M_1, M_2, M_3, M_4, M_5 }

and objects of C:

$O_1, O_2, O_3, O_4, O_5, O_6$.

Suppose the horizontal fragments of C are:

$F_{h_1} = \{ O_1, O_2, O_5 \}$;
$F_{h_2} = \{ O_3, O_6 \}$;
$F_{h_3} = \{ O_4 \}$;

and the vertical fragments of C are:

$F_{V_1} = \{O_1(v_1), O_2(v_1), O_3(v_1), O_4(v_1), O_5(v_1), O_6(v_1)\}$;
$F_{V_2} = \{O_1(v_2), O_2(v_2), O_3(v_2), O_4(v_2), O_5(v_2), O_6(v_2)\}$;

where

$v_1 = A_1, A_3, M_2, M_3$;

v_2= A_2, A_4, M_1, M_4, M_5.

The corresponding physical fragments are:

$F_{h_1} \cap F_{V_1}$= {$O_1(v_1)$, $O_2(v_1)$, $O_5(v_1)$} = {O_{11}, O_{21}, O_{51}};
$F_{h_1} \cap F_{V_2}$= {$O_1(v_2)$, $O_2(v_2)$, $O_5(v_2)$}= {O_{12}, O_{22}, O_{52}};
$F_{h_2} \cap F_{V_1}$= {$O_3(v_1)$, $O_6(v_1)$}= {O_{31}, O_{61}};
$F_{h_2} \cap F_{V_2}$= {$O_3(v_2)$, $O_6(v_2)$}= {O_{32}, O_{62}};
$F_{h_3} \cap F_{V_1}$= {$O_4(v_1)$}= {O_{41}};
$F_{h_3} \cap F_{V_2}$= {$O_4(v_2)$}= {O_{42}};

where Oi(vj) is simplified as Oij.

A graphical illustration of this example is shown in Figure 4.1, where each shaded rectangle indicates a physical fragment.

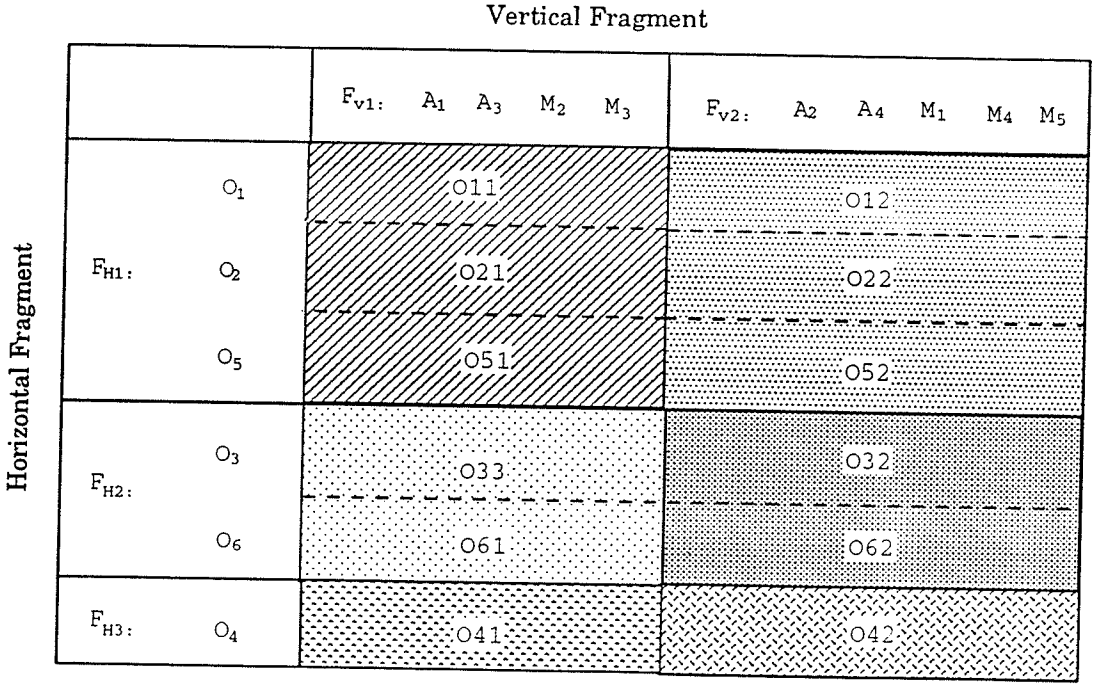


Figure 4.1 An Example of Physical Fragmentation

From the physical fragment definition and the example above, it can be seen that since vertical fragments and horizontal fragments are orthogonal to each other, every vertical fragment and horizontal fragment has only a single intersection which is a physical fragment. None of the physical fragments overlap with any other physical fragments. Each logical fragment can be uniquely represented by an ordered collection of physical fragments. According to the Definition 1.8, the 1:1 mapping can be built between hybrid fragments and physical fragments. Thus physical fragments may directly support hybrid fragments. Physical fragments are the suitable basic units of data storage. Using physical fragments, there will be no data redundancy in the store and therefore storage efficiency is assured.

4.3 Storing Physical Fragments

Since physical fragments are the unit of storage, the object store must be designed to accommodate them. To decrease implementation complexity and to exploit the features of existing code, an existing storage structure must be selected for physical fragment storage.

4.3.1 The FFS Approach

A conventional file is a storage unit. The most effective way to build a store for physical fragments is to construct it out of existing components. As discussed in Chapter

3, the Unix FFS (Fast File System) will be employed as the basic storage system. The Unix FFS was chosen for the following reasons:

- Unix is a widely used, well documented operating system and the source code of the FFS is available for this project.
- The Unix file system is flexible to use. Files can be managed either at the operating system level, where access is by inode numbers, or at the user level where pathnames are used. The inode mechanism supports efficient file access. When the object store is functioning (under Mach), access to data in the store will be through inode numbers, instead of pathnames. In this way, the performance of the store will be greatly enhanced. The pathname based naming scheme, however, will be employed when creating the object store (under Unix) to simplify the creation process.
- The FFS improves on the traditional Unix file system in several useful ways. The availability of cylinder groups is directly applicable to object storage for the purpose of grouping related fragments together and the adjustable disk layout polices can be utilized to optimally allocate fragments.
- The FFS is supported under Mach without the presence of Unix itself. The Mach code used for *reading* an existing Unix file system is already available and can be re-used.

4.3.2 Design Strategies

Given the overall goals of the design, certain specific design issues for the object

store were arrived at. The strategies of the design and implementation are based on the following decisions concerning the object store:

- The object store will associate each logical fragment with a unique fragment identifier (FID), and associate each object with a unique object identifier (OID). Fragments and objects will be referenced only through their FIDs and OIDs.
- Only non-redundant data will be stored in the object store. To do this, logical fragments need to be further decomposed into physical fragments which will be the basic data storage units in the store.
- Since objects are blended into the fragments, when objects are referenced, they need to be extracted from one or more fragments. Thus, the object store will store the information to implement mapping from any object's identifier to its locations in physical fragments in which the object is blended.
- To retrieve fragments and objects, the object store will construct an index system. The *fundamental* index structures will be kept persistent so that the object store will be robust enough to recover from system failures.
- The object store will keep the data of related fragments as physically close together as possible to decrease data access time and thus improve the performance of the system. Four kinds of related data will be considered for co-location. These are the following (in priority order):

  (1) Data in the same physical fragments,

(2) Data of those physical fragments belonging to the same logical fragments,

(3) Data of those logical fragments related to one another by membership in a given class, and

(4) Data of those logical fragments belonging to different classes but related by class inheritance.

4.4 The System Architecture

Figure 4.2 shows the system architecture. The implementation environment is the bare Mach microkernel. User applications run over the distributed shared virtual memory and access objects therein. An object store with its storage management is built on top of the Mach kernel to support object persistence in the virtual memory.

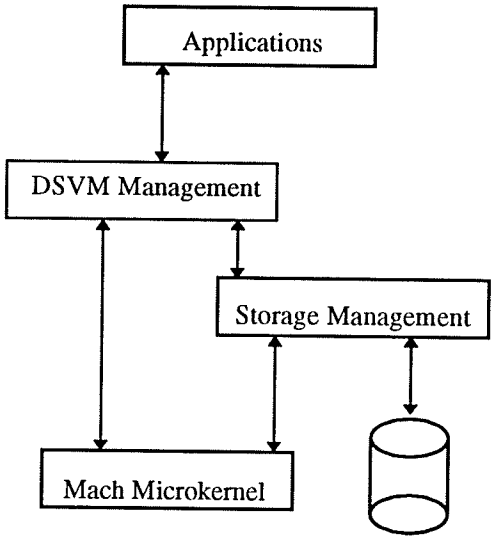


Figure 4.2 System Architecture

There are two main functions of the storage management software. The first is to create data structures for managing the fragments that reside in the object store, and the second is to handle object and fragment lookups. The internal architecture of the Storage Manager (SM) is shown in Figure 4.3.

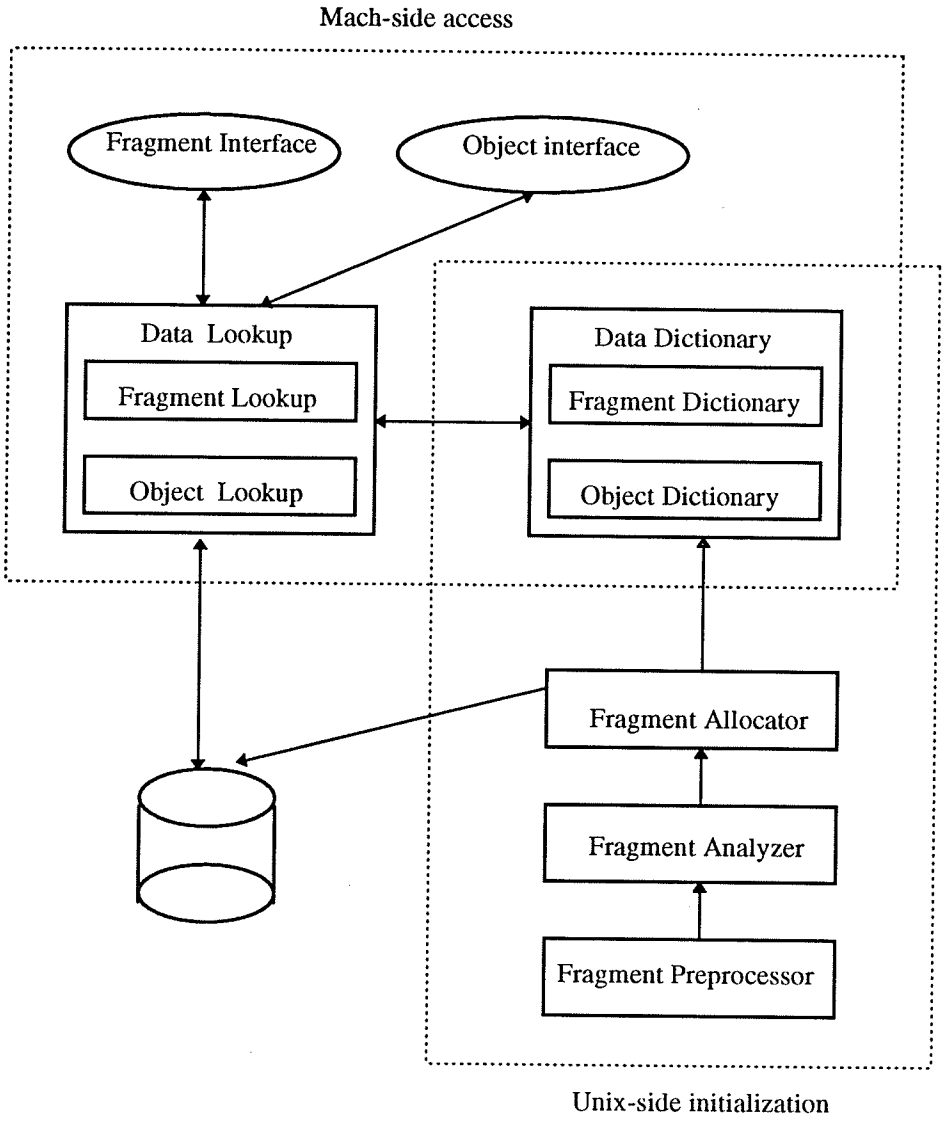


Figure 4.3 Storage Management

Each logical module of the SM is described below:

- Fragment Interface
  - Through this interface, a fragment identifier is sent to the SM, and the corresponding location of the fragment will be returned.
- Object Interface
  - Through this interface, an object identifier is sent to the SM, and the corresponding location(s) of the object will be returned.
- Data Dictionary

  This module constructs index structures for the data in the object store. It consists of two sub-modules:
  - Fragment Dictionary

    This module maintains an index structure for retrieving fragments in the store. It includes two components, a vertical fragment dictionary and a horizontal fragment dictionary.
  - Object Dictionary

    This module maintains an index structure for retrieving objects in the store.
- Data Lookup

  This module is responsible for data lookup in the object store and consists of two sub-modules:
  - Fragment lookup

This module handles searching for fragments using the Fragment Dictionary. It includes two components, a vertical fragment searcher and a horizontal fragment searcher.

- Object lookup

  This module handles searching for objects using the Object Dictionary.

- Fragment Preprocessor

  This module fetches the pre-fragmented logical fragments and prepares them for further processing by the SM.

- Fragment Analyzer

  This module analyzes the data of logical fragments sent from the Fragment Preprocessor and decomposes the logical fragments into non-redundant data units (i.e. physical fragments).

- Fragment Allocator

  This module organizes the physical fragments in terms of their relations, and attempts to allocate them onto disk blocks based on the FFS disk layout and allocation policies. The Fragment Allocator is also responsible for generating the index structures of the store for the Data Dictionary.

4.5 Layout and Allocation Policies

The layout and allocation policies of the BSD fast file system [Lef89] provide

direct support for data storage in an object store. This design tries to perform site-local fragment allocation in accordance with the requirements of the object store using the FFS layout and allocation scheme. Fragment allocation is done on Unix at the object store creation stage. Object/Fragment accesses will be processed via Mach.

There are two major allocatable resources in the FFS. The first is data blocks. The FFS layout policy routines attempt to place data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. Because of this, it is natural to map each basic data unit of the object store, a physical fragment, to an individual Unix file. This guarantees the best possible block allocation within a given physical fragment and meets the design requirement of keeping the data belonging to the same physical fragment physically close together. This is the highest priority relationship for co-location as described in Section 4.3.2 (related data, type (1)).

The second allocatable resource is inodes. The inode layout policy of the FFS attempts to place all the inodes of files in a given directory in the same cylinder group. For this reason, fragments belonging to the same class are placed in one directory at store creation time so that related data of types (2) and (3) as described in Section 4.3.2 can be kept close together. Furthermore, since the allocation policy tries to place a directory as close as possible to its subdirectories, subdirectories are used to store a class' subclasses. Thus the fragments which are related due to the inheritance relationship (related data of type (4)) will be kept close together. Figure 4.4 shows the filesystem hierarchy of the object store during store creation, where each directory corresponds to a class and each data file in each directory corresponds to a physical fragment. The inheritance hierarchy of the object store is captured in the tree hierarchy of the filesystem. Under the root

directory of the filesystem, each subtree corresponds to an inheritance relationship of the classes in the store. In Figure 4.4, for example, class(2) which is represented as a directory, is the ancestor of all the classes in that subtree. "pf(k)", represented as a file, is one physical fragment of "class(2)" which represented as a subdirectory, is a subclass of class(2), etc.

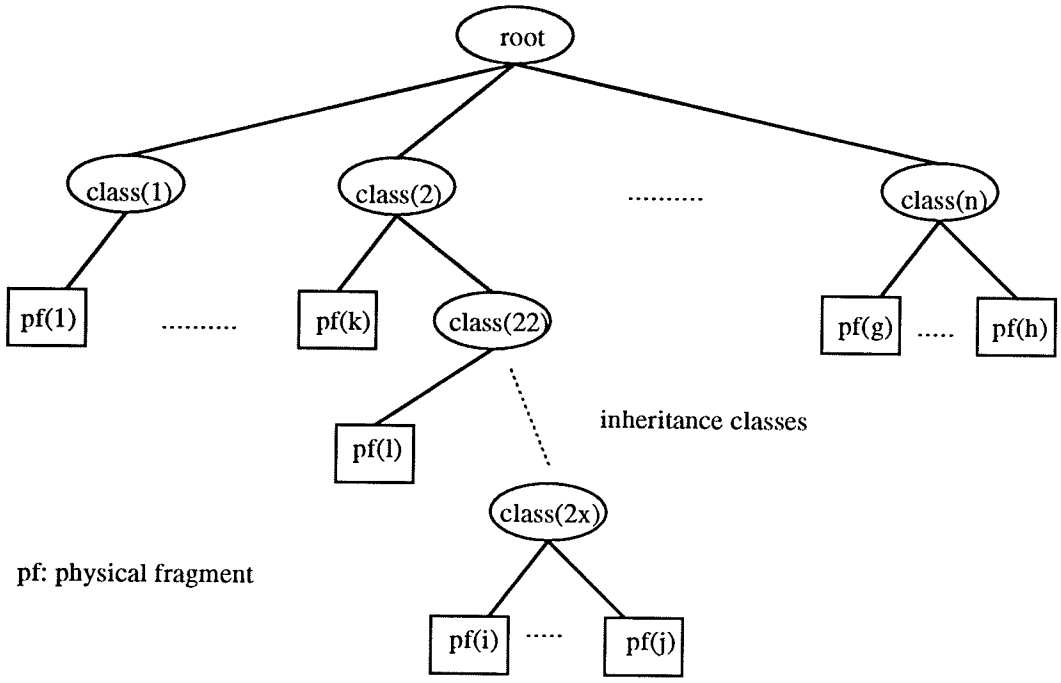


Figure 4.4 Filesystem Structure

The FFS provides a dynamic environment in which files can grow or shrink at any time. The object store, however, is currently a static system as the fragments in the store do not grow or shrink. The FFS allocation policies are still applicable in the object store for the following two reasons:

- It will make future work easier when the static store is extended to a dynamic one.
- Even in a dynamic framework, the system parameters can be adjusted to provide the best environment for a static store.

The existing policies regarding free space, such as allocating 1Mbyte worth of file blocks in each cylinder group and keeping 10% free blocks in the filesystem are still relevant [Lef89]. When a new directory needs to be allocated, a cylinder group with a greater than average number of unused inodes will be chosen. With these strategies, the system should be able to keep related fragments together in the same cylinder group even after they are expanded because of modifications. Further details of existing policies and their application to the storage of fragments and objects are now presented.

The FFS has two levels of disk block allocation routines which are referred to as the global and local policy routines. As applied in the object store, the global policy routines will try to select blocks in the same cylinder groups for sufficiently small fragments. For larger fragments, when further block allocation must be redirected to a different cylinder group, each newly selected block is chosen from the nearest cylinder group having more than average free space.

The global policy routines call local allocation routines for specific blocks. The local allocation routines will always allocate the fragments if the requested block is free. If not, the local allocator applies the following allocation strategy [Lef89]:

(1) Use the next available block rotationally closest to the requested block on the same cylinder.

(2) Use a block within the same cylinder group, if (1) fails.

(3) Quadratically hash the cylinder group number to choose another cylinder group with a free block, if (2) fails. Quadratic hashing is used because of its speed in finding unused slots in nearly full hash tables [Lef89, Knu75].

(4) Apply a search to all cylinder groups for a free block, if (3) fails.

A problem existing in the layout policies of inodes and data blocks is that for large fragments, free space may easily be filled. This problem may be partially addressed within the existing FFS by:

- reducing the number of inodes (to make more room for data); or
- increase the size of the blocks (to decrease fragmentation).

These must be done during the filesystem's creation.

4.6 Data Access

Object and fragment identity are the foundation of data referencing in the store. An OID/FID is an invariant property which logically and physically distinguishes one object/fragment from another. Data search structures in the object store include object/fragment identifiers at both logical (conceptual schema) and internal (physical schema) levels, in addition to the mapping schemes between the two.

4.6.1 Data Mapping

Logical references to data in the object store are by logical fragment identifiers,

represented by FIDs, and by object identifiers, represented by OIDs. FIDs and OIDs are the starting point for accessing data.

Since logical fragments have been further decomposed into physical fragments which are the basic data units on disk, every vertical fragment or horizontal fragment corresponds to a set of one or more physical fragments. Therefore, the internal notion of each logical fragment, the internal fragment identifier, is represented as a set of inode numbers of the files containing the physical fragments.

The internal notion of objects (the internal object identifier) is more complicated than that of logical fragments since objects are blended into fragments. A physical fragment may contain only a part of an object and it may contain one or more such parts of several different objects. Thus in addition to the inode numbers of physical fragments containing the parts of an object, an internal object identifier needs to map to information about the offsets and lengths of each object segment in the relevant physical fragments. Data access ends when the internal object/fragment identifiers are mapped and the required bytes (or data location(s)) are returned.

Searching for a logical fragment or an object is the process of mapping a FID or an OID to its internal notation:

FID $\Rightarrow$ { i-numbers of the physical fragments}

OID $\Rightarrow$ {i-numbers of the physical fragments, the object segment offsets and the segment lengths in the physical fragments}.

4.6.2 Index Structures

To realize the mapping from FIDs (or OIDs) to the relevant locations, certain

index structures are required. Such index structures maintain the inter-relationships among objects and fragments. They serve as FID and OID dictionaries as shown in the architecture of the SM (Figure 4.3) and provide efficient data search for the object store. B^+-trees, together with certain supporting index tables are used as the data structures for implementing the FID and OID dictionaries.

B^+-trees are a widely used data structure for organizing and maintaining large indices. They are designed to support the operations: *create, insert, delete* and *lookup*. For many years, the B^+-tree has been the data structure of choice for applications requiring both sequential and direct access. Among the B-tree and its variants, the B^+-tree is most suitable to serve as the index structure for the object store. The advantages of using the B^+-tree are discussed in [Gra93, Joh93]. Since the internal nodes in a B^+-tree do not contain data pointers, more tree pointers can be packed in each node. For the same size of disk block, a B^+-tree supports a larger order tree than a B-tree. This leads to fewer tree levels, and thus shortens tree search time.

Another popular data structure which is good for disk storage is extendible hashing. If the database is *very* large, extendible hashing may be used. If the database is not that large, a B^+-tree may be used. The B^+-tree was chosen for several reasons. As long as the root page of the B^+-tree is cached in memory, the performance of the B^+-tree is at least as good as extendible hashing [Kim90b], if not better. B^+-trees also permit lookup with testing for key ranges rather than only exact key matches as with hashing schemes. Finally, the B^+-tree is easily partitioned for use in a distributed environment.

Many important object oriented databases use the B-tree and its variants to map the object identifiers to their physical addresses. For example, GemStone and EXODUS support a B^+-tree index on objects. Statice uses a B*-tree, Iris uses a B-tree, and O2 uses a "B-tree like ordered tree" to access methods. Although ORION uses extendible hashing, a B^+-tree is also used to speed up the associative searches for objects. In future improvements to this object store, a B^+-tree combined with hashing methods might be applied to achieve the best performance in lookup operations.

The existence of index structures after system crashes is an important feature of persistence. To assist fast recovery, the important parts of the index structures will be stored on the stable disk (i.e. "persistently") so that they can be used to reset the data retrieval structures after software failures[1]. Such index structures can be regarded as another type of persistent object, which might be called persistent index objects. Reusing these persistent index objects provides an effective approach to object store protection regarding software failures. In future, a logging approach might be also considered to support object store recovery from hardware failures.

[1] Note that this in no way precludes in-memory caching to improve performance.

Chapter 5

Implementation

The object store is implemented as two major components. The first component is Object Store Creation (OSC) which occurs under Unix. The second component is Object Store Access which occurs under Mach. The input data of this implementation is based on the vertical and horizontal fragment output results of Ezeife and Barker [Eze94a, Eze94b, Eze95].

5.1 Object Store Creation

The task of Object Store Creation is to determine the logical data structures and disk data format required for the object store. The process of Object Store Creation is shown in Figure 5.1. The system first loads the pre-fragmented logical fragments (vertical and horizontal) into memory where they are further decomposed into physical fragments

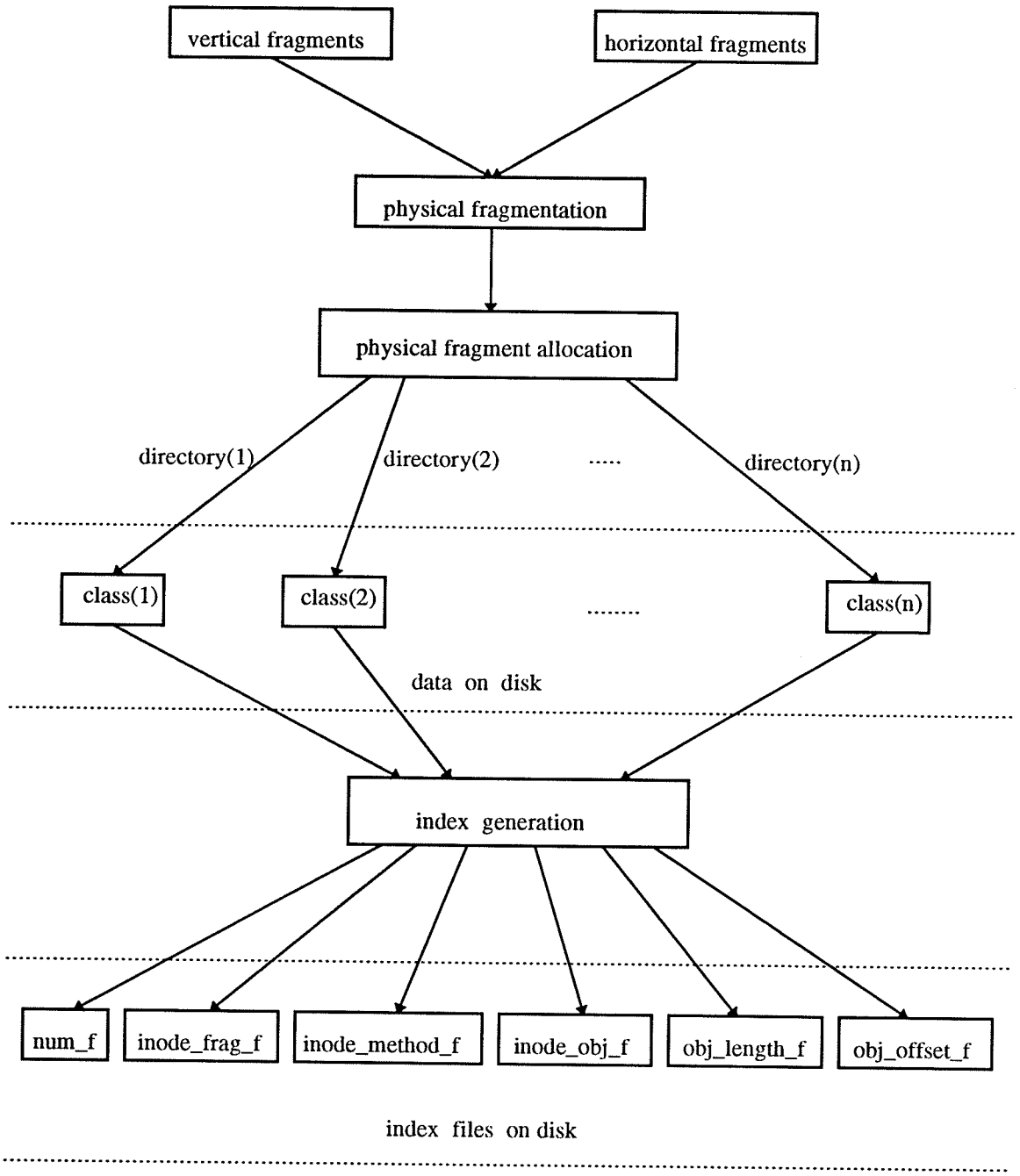


Figure 5.1 Object Store Creation

to generate non-redundant data clusters for storage. These non-redundant physical fragments, are then allocated to disk using the FFS. Finally the system creates persistent indexes for the purpose of data retrieval.

5.1.1 Data Structures

The logical data structures for each class used to implement the object store include the following:

- Data structures supporting vertical fragments
  - *v_frag { *attr, *method}*

    v_frag represents a vertical fragment. It contains two pointers **attr* and **method*. **attr* points to a vertically fragmented object vector, which contains a byte sequence of one or more attributes of the class. **method* points to a method vector for the same vertical fragment of the class[1].
  - *v_frag_class { v_frag_num, v_frag }*

    v_frag_class represents a vertically fragmented class. It has two fields *v_frag_num* and *v_frag*. The *v_frag_num* field represents the number of vertical fragments in the class. The *v_frag* field represents each vertical fragment in the class.
- Data structure supporting horizontal fragments
  - *h_frag_class { h_frag_num, *h_obj_order, *h_obj_num }*

[1] The contents of **method* is the set of file names of the files containing the methods for each vertical fragment.

h_frag_class is the format of a horizontally fragmented class. It contains information about the horizontal fragmentation of a class including *h_frag_num, *h_obj_order* and **h_obj_num. h_frag_num* indicates the number of horizontal fragments in the class. **h_obj_order* is a pointer to a vector which specifies the order of objects in each horizontal fragment of the class. **h_obj_num* is a pointer to a vector which specifies the number of objects in each horizontal fragment of the class.

- Data structures supporting physical fragments
  - *p_frag { *attr, *method}*

    p_frag represents a physical fragment of a class. It consists of two pointers **attr* and **method. *attr* points to a physically fragmented object vector. **method* points to a method vector in the same physical fragment of the class[2].

  - *p_frag_class { p_frag_num, p_frag }*

    p_frag_ class represents a physically fragmented class. It has two fields *p_frag_num* and *p_frag*. The *p_frag_num* field stands for the number of physical fragments in the class. The *p_frag* field stands for each physical fragment in the class.

- Data structure used in every class
  - *obj_num*

    obj_num indicates the number of objects in the class.

[2] The content of **method* of *p_frag* is the same as that of *v_frag*, but the disk location is changed.

v_frag_class, including all corresponding *v_frags*, and *h_frag_class* are input variables. The vertical fragments of all classes are assumed to exist in a file (*v_file*). As horizontal fragmentation is done on the data of the same classes, there is no need to keep the horizontal fragment data (since it occurs within the physical fragments corresponding to the vertical fragments). Therefore, only the *format* (refer to Figure 5.2 which will be discussed later) of horizontal fragmentation, *h_frag_class*, is required. In this way, storage is saved and data copying operations are reduced. It is assumed that the *h_frag_class* information for all classes is available in a file (*h_file*) along with *obj_num* and the total number of classes. It is further assumed that the code for methods belonging to each vertical fragment are available in individual temporary files for each class. All these temporary file names for the methods of all classes are kept in a file (*m_file*). This assumption makes the object store implementation simpler, since when the final location of the method code is determined on disk, the only thing that needs to be done is to change the file name and delete the temporary files. The time-consuming work of transferring the bytes of method codes to the destinations is eliminated. These assumptions are reasonable because the fragmentation is done statically. Things can be easily arranged this way. When the system finishes physical fragmentation and generates the output variables *p_frag_class* (including all *p_frag* structures) the pre-determined input files (*v_file*, *h_file* and *m_file*) will be deleted.

5.1.2 The Algorithm

The algorithm for object store creation consists of four steps, FetchFrag, PhyFrag,

StoreFrag, and StoreIndex. The algorithm is shown below and discussed in detail in Section 5.1.3. The symbol "-->" used in the following algorithm indicates copying.

Algorithm ObjectStoreCreation

begin

for *each class* ***do***

Step one: // FetchFrag gets the vertical fragments of the current class, v_frag_class, data concerning the horizontal fragments of the class, h_frag_class, and object numbers of the class, obj_num. //

v_file --> v_frag_class;

h_file --> h_frag_class;

input obj_num

Step two: // PhyFrag does physical fragmentation. //

for *each v_frag ∈ v_frag_class* ***do***

Sort the object parts in v_frag into the same order as in the horizontal fragments as specified by h_obj_order;

Divide the sorted vertical fragment v_frag into segments reflecting the horizontal fragmentation using h_obj_num;

Generate physical fragments p_frag;

end;

Step three: // StoreFrag stores the physical fragments and their inode-numbers onto the disk. //

Generate a pathname for the current class and create a directory with that pathname for the fragments of the class;

for *each p_frag ∈ p_frag_class* ***do*** *// handle physical fragments //*

Generate a pathname and create a file within the directory;

**attr of p_frag --> the created file*

end;

for *each v_frag ∈ v_frag_class* ***do*** *// handle methods belonging to the physcal fragments //*

Generate a pathname within the directory;

*Link *method of p_frag with the pathname*

end;

Get the inode-numbers of all physical fragments and methods, and store them in a file phy_inode_f.

Step four: // StoreIndex stores the index information for fragments and objects. //

```
    v_frag_num, h_frag_num  and obj_num --> num_f;
    for each p_frag ∈ p_frag_class do    // store the inode numbers of physical fragments //
        Pick up the inode-number of *attr in p_frag from phy_inode_f;
        The inode-number --> inode_frag_f
    end;
    for each v_frag ∈ v_frag_class do   // store  the inode numbers of methods //
        Pick up the inode-number of *method in p_frag from phy_inode_f;
        The inode-number --> inode_method_f
    end;
    for each object ∈ p_frag_class do   //store  objects' inode numbers, lengths and offsets//
        Search and collect the inode-numbers of the object segments from phy_inode_f;
        The inode-numbers --> inode_object;
        The size of the object --> length;
        offset + length  --> offset;
        inode_object, length and offset --> inode_obj_f, obj_length_f and obj_offset_f
    end;
  end;
end.
```

The logical fragment identifiers, FIDs, are assigned by the order of logical fragment input. This scheme is simple for generating FIDs, and it also guarantees that each FID is unique because it assumes that each logical fragments is input only once. Consider the following example:

Class No. 1

Vertical fragments: v1, v2, and v3;	FID(v1)=1, FID(v2)=2, and FID(v3)=3.
Horizontal fragments: h1, h2;	FID(h1)=1, and FID(h2)=2.

Class No. 2

Vertical fragments: v1, and v2;	FID(v1)=4, and FID(v2)=5.

Horizontal fragments: h1, and h2; FID(h1)=3, and FID(h2)=4.

The same scheme is used for object identifiers, OIDs. The object's position in the first vertical fragment of a class is used as the object's identifier. This number increases contiguously for the classes one by one. If, in the future, a more user friendly FID/OID is desired, say a literal FID/OID, a map may be built between the literal FID/OID and the corresponding numeric FID/OID.

5.1.3 Algorithm Discussion

The first step of Object Store Creation is to fetch the logical fragments, from files *v_file*, *h_file* and *m_file* into memory. OSC treats the fragment data as sequences of bytes. As the data is loaded, OSC sets up the corresponding data structures in memory. These include *v_frag_class*, and *h_frag_class*. By this process the data is placed in a form upon which OSC can start the physical fragmentation.

The second step is to decompose the logical fragments into physical fragments. The main task of this step is to generate a new data structure, *p_frag_class,* based on *v_frag_class* and *h_frag_class*. Since both vertical and horizontal fragmentation act on the same class, the decomposition into physical fragments needs two steps as follows:

- Rearranging the data in the class.
- Determining the intersection of vertical and horizontal fragments in the class.

To do this, OSC first rearranges the order of the objects in each vertical fragment according to the object identifier in each horizontal fragment in the same class. Then OSC decomposes each reordered vertical fragment into a number (*h_frag_num*) of portions. The number of object segments in each portion is the same as the number of objects in each horizontal fragment in the same class. Finally, these portions are generated as physical fragments in *p_frag_class*. As an example, Figure 5.2 illustrates the process of generating physical fragments.

The third step is to allocate physical fragments to disk using the FFS. Each class corresponds to a directory. All the physical fragments belonging to the same class are stored in the same directory. In a directory, the attributes belonging to the same physical fragments are assigned to the same file. Each vertically fragmented method of the class is assigned to an individual file in the directory. Classes are numbered by their order of appearance in the input sequence. For implementation simplicity, a directory pathname is constructed simply using the class' number. A physical fragment file name is the directory pathname suffixed with the number of the physical fragment in the fragment sequence of the class. A method file name is the directory pathname suffixed with the number of the method in the vertically fragmented method sequence of the class. Once all physical fragments are assigned to disk, OSC records the inode numbers of the physical fragments and method files in a temporary file, *phy_inode_f*, which will be used to construct the index files for data retrieval in the next step, Object Store Access (OSA).

If class inheritance relationships are considered, the object store will have a tree (hierarchical) structure. In this implementation, however, all classes are treated equally.

In a given class C:

The objects of C: $\{O_1, O_2, O_3, O_4, O_5\}$;

The vertical fragments of C: $\{v_1, v_2, v_3\}$; The horizontal fragments of C: $\{h_1, h_2\}$.

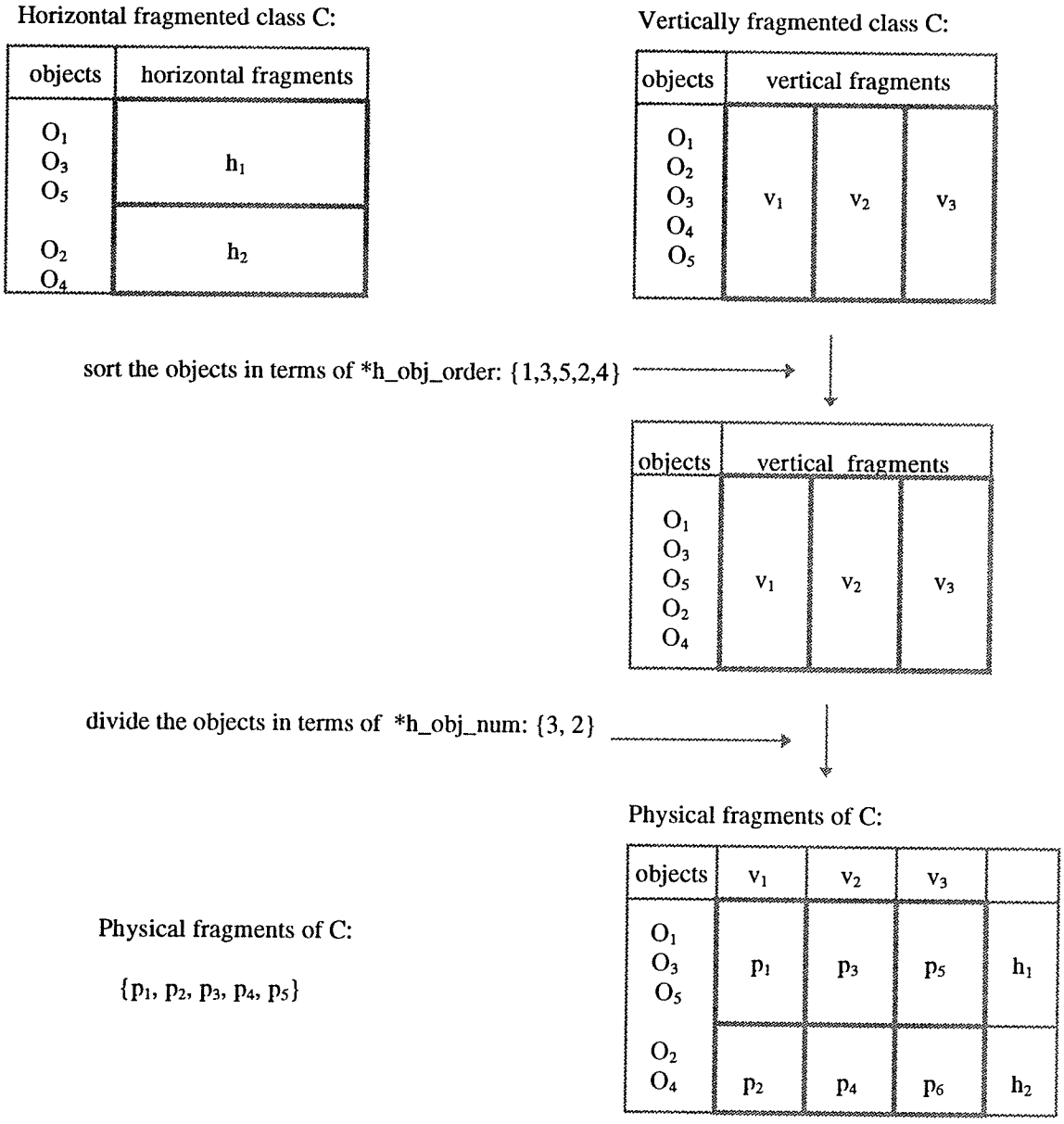


Figure 5.2 The Procedure of Physical Fragmentation

Thus, under the root directory there is only one level of subdirectories. The implementation support of class inheritance will be left for future work.

The last step of OSC is to determine and construct the smallest possible index for the object store's contents. These indexes must be persistent with the store so that the system retrieval structure, which is built dynamically, can be recovered quickly after the system crashes. Once the index files for all classes are created, pathnames are no longer used. Accessing fragments and objects is done using their inode numbers, instead of their file names.

The first index file is *num_f* which contains the number of vertical fragments, the number of horizontal fragments and the number of objects in each class. In the implementation of Object Store Access, these numbers will be used to build index tables and support operations on the tables.

Based on *phy_inode_f*, OSC determines the inode numbers of physical fragments and methods, and fills them in the index files *inode_frag_f* and *inode_method_f* respectively. For objects, however, more indexes are needed because an object is not an independent storage unit. Since each object may be blended in to multiple physical fragments, OSC checks which inode numbers the object is stored in and picks them up from the data set of physical fragment inode numbers. In addition, OSC needs to determine the lengths and the offsets of the object segments in the physical fragments. File *inode_obj_f* contains the inode numbers of the physical fragments containing objects. Files

obj_length_f and *obj_offset_f* specify object lengths and object offsets within the physical fragments respectively[3]. An object's identifier is used to index into these tables.

5.2 Object Store Access

The task of Object Store Access (OSA) is to create index tables and data retrieval structures in the object store and then support object and fragment access using them. For logical fragments, given a FID, the system will return a set of inode numbers by which the logical fragment can be accessed[4]. For objects, given an OID, the system will also return a set of inode numbers. In addition to each inode number there are two associated numbers. One indicates the length of the object segment and the other one indicates the offset of the object segment in the corresponding physical fragment.

5.2.1 Index Structures

The OSA has a three-level index structure (as shown in Figure 5.3). In the first level, there are six tables, storing information for all classes, as follows:

- *num_table*: the table of *v_frag_num*, *h_frag_num* and *obj_num data*;
- *inode_frag_table*: the table of inode-numbers of physical fragments;
- *inode_method_table*: the table of inode-numbers of methods;

[3] The reason to use three files instead of one is to make the concepts clear and consistent between Object Store Creation section and Object Store Access section. In the latter section, three tables, *inode_obj_table*, *obj_length_table*, and *obj_offset_table* are needed.

[4] As implemented, OSA returns locations of data rather than the data itself. Access is made using the existing FFS 'read', 'write', and 'seek' operations. Returning the actual data would require only simple modifications to the code.

- *inode_obj_table*: the table of inode-numbers of objects;
- *obj_length_table*: the table of object segment lengths within inodes;
- *obj_offset_table*: the table of object segment offsets within inodes..

The structures implementing the second level index tables (for all classes) are as follows:

- *frag_l2_index* { *num, inode_frag, inode_method*}

  frag_l2_index stands for a second level index table for logical fragments. Each entry of *frag_l2_index* contains three fields, *num, inode_frag,* and *inode_method,* associated with each class. For any given class, *num* points to its *num_table*; *inode_frag* points to its *inode_frag_table*; and *inode_method* points to its *inode_method_table.*

- *obj_l2_index* { *num, inode_attr, inode_method*}

  obj_l2_index stands for a second level index table for objects. Each entry of *obj_l2_index* contains three fields, *num, inode_attr,* and *inode_method,* associated with each class. For a given class, *num* points to its *num_table*; *inode_obj* points to its *inode_obj_table,* (along with its *obj_length_table* and *obj_offset_table*) and *inode_method* points to its *inode_method_table.*

The third (top) level index structures are B^+-trees. In the B^+-tree [Smi87, Elm89], data pointers are stored only at the leaf nodes of the tree. Each data pointer points to an entry in a second level index table. Key pointers are used to search through the B^+-tree based on FID/OID to locate the lower level indexes' structures describing the

corresponding fragment/object. The other type of pointers, tree pointers, connect the tree nodes to enable lookups.

5.2.2 Index Table Setup

Based on the data in the six original tables, creating index tables for the object store involves filling in index table entries at all three levels. Figure 5.3 shows the layout of the three level index structure.

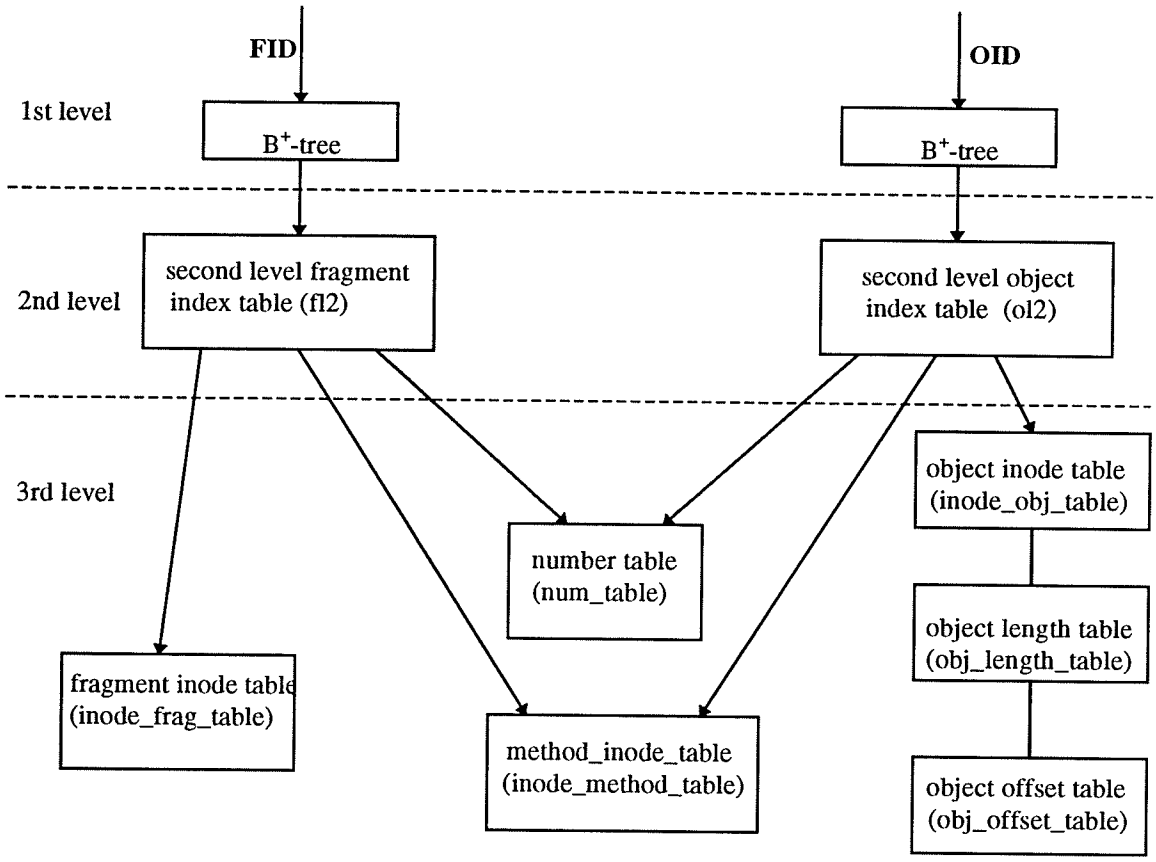


Figure 5.3 Index Tables of the Object Store

The first step in setting up the index tables is to fetch the persistent data in the index files written during OSC and store them in the first level index tables. The six index files, *num_f, inode_frag_f, inode_method_f, inode_obj_f, obj_length_f,* and *obj_offset_f* correspond to the six first level index tables, *num_table, inode_frag_table, inode_method_table, inode_obj_table, inode_length_table,* and *inode_offset_table.* The algorithm SetupTables will fill the six index tables with the data provided by the six index files. Again, the symbol "-->" indicates copying in the following algorithms.

Algorithm SetupTables

for *all classes* ***do***

v_frag_num, h_frag_num and obj_num from num_f (on disk) --> num_table (in memory);

inodes of p_frag_class from inode_frag_f (on disk) --> inode_frag_table(in memory)

inodes of vertical fragment methods from inode_method_f (on disk) --> inode_method_table (in memory)

inodes of object segments from inode_obj_f (on disk) --> inode_obj_table (in memory)

object segment lengths from obj_length_f (on disk) --> obj_length_table (in memory)

object segment offsets in each physical fragment from obj_offset_f (on disk) --> obj_offset_table (in memory)

end.

The next step is to build the second level index tables on top of the first level. Each entry in the second level index tables points to a certain position in the first level index tables. There are three second level index tables, the second level index table of vertical fragments *vfl2*, the second level index table of horizontal fragments, *hfl2*, and the second level index table of objects, *ol2*. The algorithm L2Index.vertical_fragment is used to setup *vfl2*.

Algorithm L2Index.vertical_fragment

```
const next-data-piece-location =1;
const num-table-uni t= 3;
for each class do
  for each vertical fragment do
    index.num_table --> num;
    index.inode_frag_table --> inode_frag;
    index.inode_frag_table + h_frag_num --> index.inode_frag_table;
    index.inode_method_table --> inode_method;
    index.inode_method_table + next-data-piece-location --> index.inode_method_table;
  end;
  index.num_table + num-table-unit --> index.num_table;
end
```

To setup *vfl2* three first level index tables, *num_table, inode_frag_table* and *inode_method_table*, are needed. Figure 5.4 shows the layout of the vertical fragment index structure. Each data pointer in the B^+-tree leaf level is associated with a vertical fragment, and thus is associated with three entries of *vfl2: num, inode_frag* and *inode_method.* And each of these entries points to one of the first level index tables. The *inode_frag_table* contains the inode numbers of physical fragments which are sorted as vertical fragments' order for each class. For instance, the physical fragments' inode numbers of the first vertical fragment of a class are contiguously arranged in the table, then they are continued by the inode numbers of physical fragments belonging to the second vertical fragment. Such procedure continues until the last physical fragment of the last vertical fragment of the class. Right after that, *inode_frag_table* will arrange the next class' physical fragment inode numbers with the same scheme. The inode number of methods are also arranged in *inode_method_table* in terms of the vertical fragment order

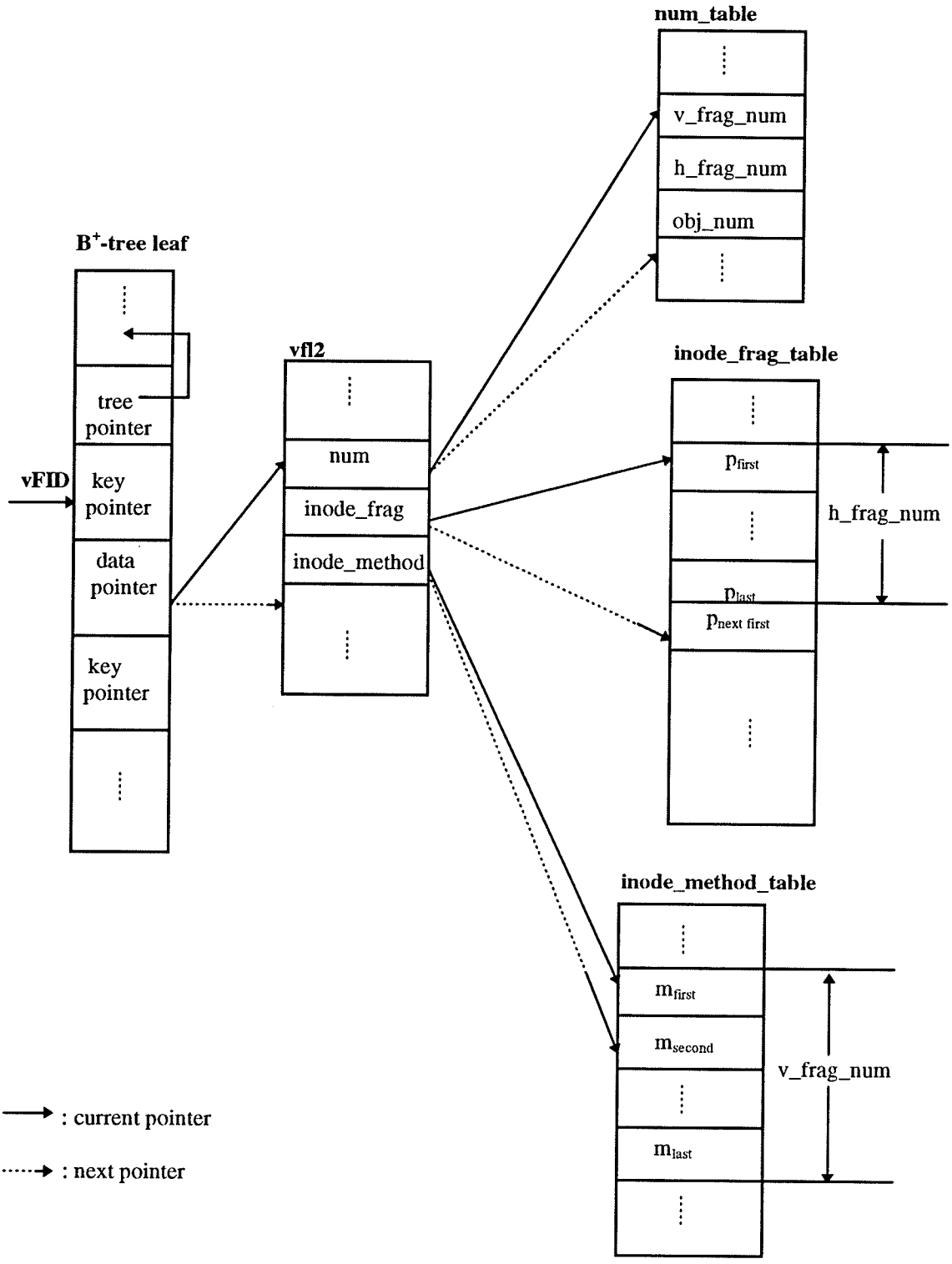


Figure 5.4 Vertical Fragment Index Tables

in every class. Each class is associate with three entries of *num_table: v_frag_num, h_frag_num,* and *obj_num.* In Figure 5.4, each solid arrow points to the certain location of another table for the current entry, while each dashed arrow is used to indicate where a certain following entry with the same type will point to. For each vertical fragment, *num* is set to the *index* of *v_frag_num* for thecurrent class in *num_table.* Thus, all vertical fragments for a given class have the same value of n*um.* For the vertical fragments in the next class *num* will point to the next *v_frag_num* entry in the table *num_table.*

In *vlf2*, the *inode_frag* entry of each vertical fragment only records the index of the first physical fragment belonging to the vertical fragment. Based on this first index and the data provided by *num*, the rest of the physical fragments can be located by offset since the indexes of the physical fragments making up a vertical fragment are stored consecutively in *inode_frag_table.* This scheme avoids the complexity of a second index table structure, and thus reduces the complexity of searching for data. The same scheme is also used in *hfl2* and *ol2. As* shown in Figure 5.4, the *inode_frag* for the first vertical fragment of the current class is set to the index of the first physical fragment (p_{first}) in that vertical fragment. Beginning from p_{first}, there are '*h_frag_num*' physical fragments which belong to the current vertical fragment. The *inode_frag* of the next (second) vertical fragment will point to the first physical fragment of that vertical fragment, $p_{next\ first}$.

The *inode_method* of the first vertical fragment of the current class is set to the index of the first method, m_{first}, in *inode_method_table.* In this table, there are '*v_frag_num*' vertically fragmented methods for the current class. The *inode_method* of

the next vertical fragment will point to the second method, m_{second} , of the class. Following these rules, *vfl2* will be setup for all classes.

Although *hfl2* shares the same tables as *vfl2*, *num_table, inode_frag_table* and *inode_method_table*, the entries of *hfl2* point to different positions in those tables. Therefore, it is necessary to setup an independent second level index table for horizontal fragments. The algorithm L2Index.horizontal_fragment is used to setup *hfl2*[5].

Algorithm L2Index.horizontal_fragment

```
const next-data-piece-location =1;
const num-table-unit= 3;
for each class do
  for each horizontal fragment do
    index.num_table -->num;
    index.inode_frag_table --> inode_frag;
    index.inode_frag_table + next-data-piece-location --> index.inode_frag_table;
    index.inode_method_table --> inode_method;
  end;
  index.num_table + num-table-unit --> index.num_table;
  index.inode_frag_table + (v_frag_num-1) * h_frag_num --> index.inode_frag_table;
  index.inode_method_table + v_frag_num --> index.inode_method_table;
end.
```

In *hfl2*, the setting of *num* entries follows the same rule as that in *vfl2*. Figure 5.5 shows the layout of the horizontal fragment index structure. The *inode_frag* field of the current horizontal fragment points to a physical fragment in *inode_frag_table,* which is the first physical fragment belonging to this horizontal fragment. The next physical fragment belonging to this horizontal fragment can be located by skipping *'h_frag_num'* physical

[5] *v_frag_num, h_frag_num* and *obj_num* in OSS are the same as they are in OSC.

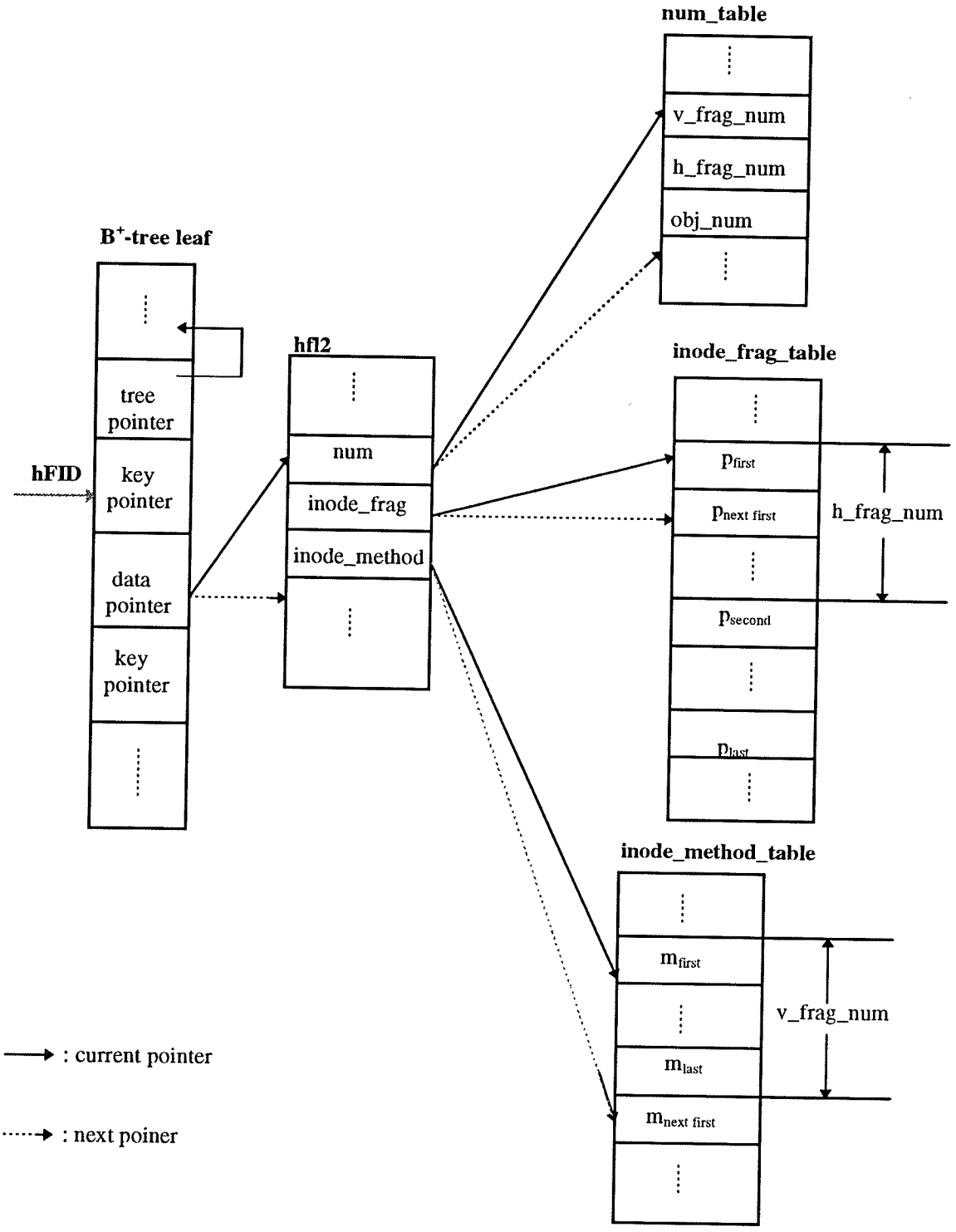


Figure 5.5 Horizontal Fragment Index Tables

fragments. Each horizontal fragment, as mentioned above, also records only the index of the first physical fragment belonging to this horizontal fragment. For the same reason, the *inode_method* of each horizontal fragment just records the index of the first method of the current class. Thus, the *inode_method* field of every horizontal fragment points to the same position in *inode_method_table* for the same class. Each horizontal fragment owns all the vertically fragmented methods of the class, from m_{first} to m_{last} in *inode_method_table*. If the position of m_{first} is available, the next following methods will be continually accessible via offsets.

The algorithm L2Index.object is used to setup a second level index table for objects.

Algorithm L2Index.object

```
const num-table-unit= 3;
for each class do
  for each object do
     index.num_table --> num;
     index.inode_obj_table --> inode_obj;
     index.inode_obj_table + v_frag_num --> index.num_table;
     index.inode_method_table --> inode_method;
  end;
     index.num_table + num-table-uni t --> index.num_table;
     index.inode_method_table + v_frag_num --> index.inode_method_table;
end.
```

To construct *ol2*, three first level index tables, *num_table, inode_obj_table*, and *inode_method_table* are needed. Figure 5.6 shows the layout of the object index structure. Since the tables, *inode_obj_table, obj_length*_table and *obj_offset_table* have the same indexes, there is no need to record *obj_length*_table and *obj_offset_table* in *ol2*. Once an

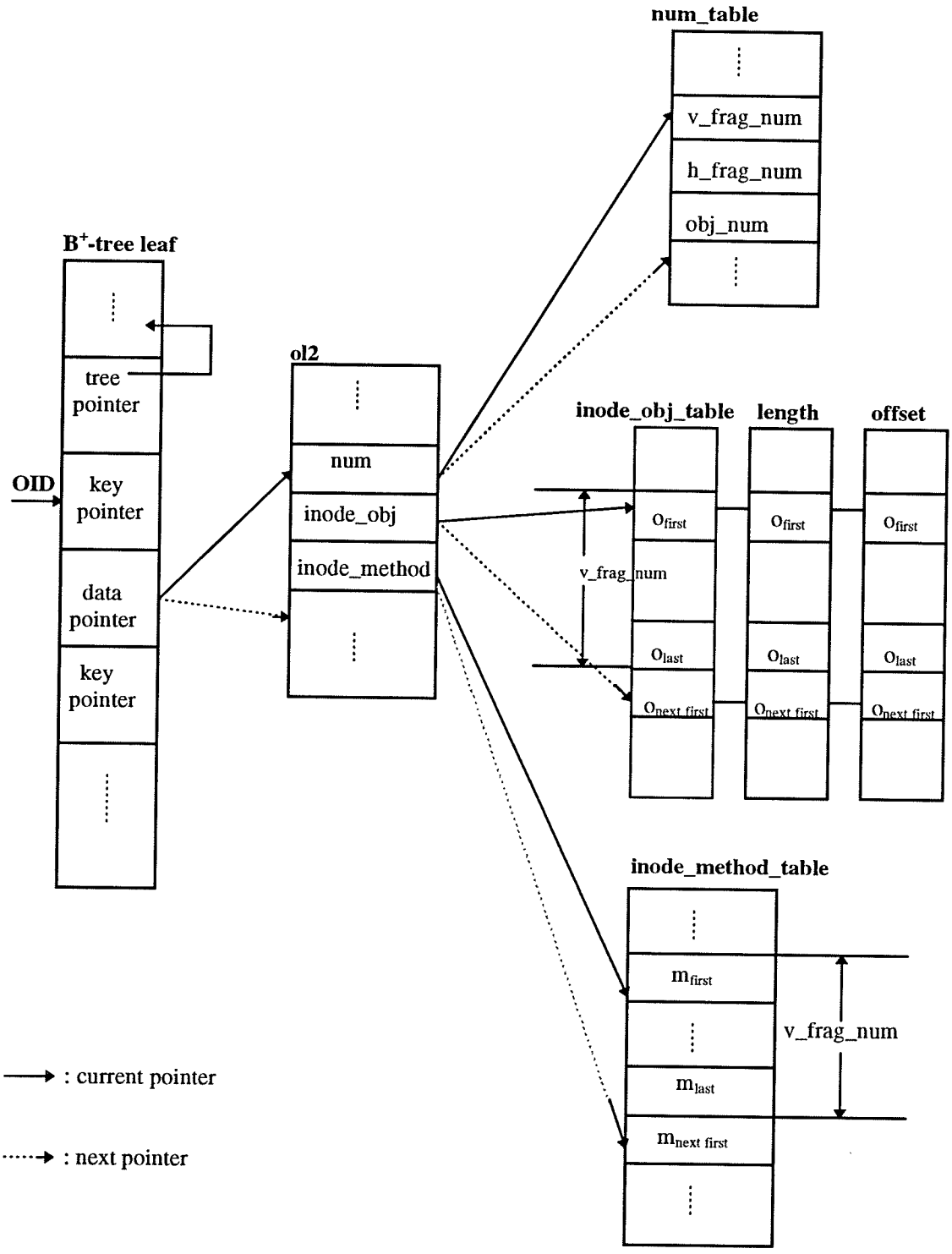


Figure 5.6 Object Index Tables

index of *inode_obj_tabl* is available, using the same index, the required data in *obj_length*_table and *obj_offset_table* can be obtained.

In *ol2*, again, setting *num* follows the same rules used in *vfl2* and *hfl2*. Therefore, for the three second level index tables, *num* is the same. There are two reasons that this element is not factored out of the three tables and made independent. One is that keeping *num* in each table makes the implementation logically clearer and also easier. The other is that this scheme makes the three kinds of searching independent. Thus any fault occurring in one table will not affect data searching in the other tables.

Each *inode_attr* in *ol2* is set to a certain index of *inode_obj_table*. Beginning from this index (o_{first}), *'v_frag_num'* contiguous inode-numbers all belong to this object. The *inode_attr* of the next object will be set to the first following index, the index of $o_{next\ first}$. That is, each *inode_attr* points to the first element of every *v_frag_num* elements in *inode_obj_table*. Setting *inode_method* in *ol2* applies the same rule as that in *hfl2*.

Finally, three B^+-trees will be built individually on each of the three second level index tables to implement the top index structure for searching logical fragments (vertical and horizontal) or objects. The algorithm FillBtree will setup the B^+-trees and connect them to *vfl2*, *hfl2*, and *ol2*.

Algorithm FillBtree

Create a B^+-tree for vertical fragments;
for *each vertical fragment of all classes* ***do***
  vertical fragment ID --> key pointer in a B^+-tree leaf node;
  frag_l2_index of vertical fragments --> data pointer in the same B^+-tree leaf node;
end;
Create a B^+-tree for horizontal fragments;

```
for each horizontal fragment of all classes do
    horizontal fragment ID --> key pointer in a B+-tree leaf node;
    frag_l2_index of horizontal fragments --> data pointer in the same B+-tree leaf node;
end;
Create a B+-tree for objects;
for each object of all classes do
    object ID --> key pointer in a B+-tree leaf node;
    obj_l2_index --> data pointer in the same B+-tree leaf node;
end.
```

Through the process of insertion (abstracted in Algorithm FillBtree), the B^+-tree key pointers are linked with FIDs/OIDs, and the data pointers with the indexes of *num* in the three second level index tables (See Figure 5.4, Figure 5.5, and Figure 5.6). Two examples are provided as follows to illustrate how the index system work.

Example 1.

Take the example of Figure 4.1. There is one class with 6 objects (O_1, O_2, O_3, O_4,O_5, O_6), 3 horizontal fragments (F_{h1}, F_{h2}, F_{h3}), 2 vertical fragments (F_{v1}, F_{v2}), and 6 physical fragments (p_1, p_2, p_3, p_4, p_5, p_6). Figure 5.7 shows the logical and disk views of the fragmented class. Its vertical fragment index tables and object index tables are given in Figure 5.8 and Figure 5.9 respectively.

Example 2.

This example illustrates the situation with two classes (C_1, C_2). Class C_1 has 3 objects and is only horizontally fragmented into 2 horizontal fragments (F_{h1}, F_{h2}). Thus C_1 can only have 2 physical fragments (p_{11}, p_{12}) which are the same as the two horizontal

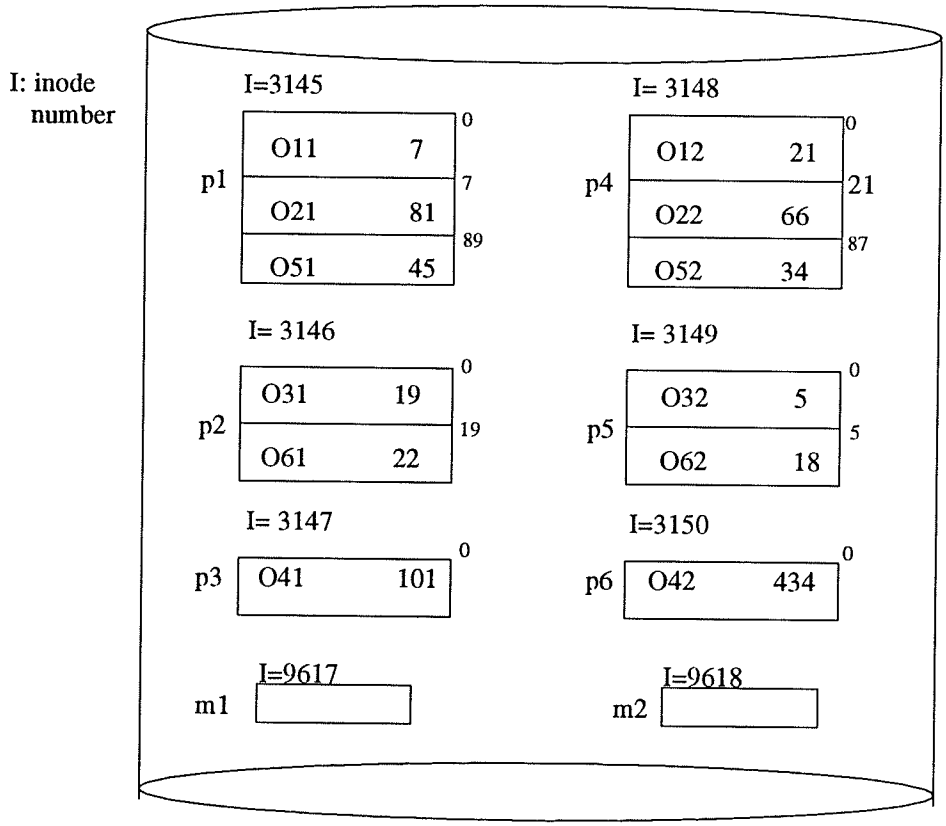
	F_{v1}	F_{v2}
F_{h1}	p_1	p_4
F_{h2}	p_2	p_5
F_{h3}	p_3	p_6
method	m1	m2

(a)

		F_{v1}	F_{v2}
F_{h1}	O_1	O11: len=7, off=0	O12: len=21, off=0
	O_2	O21: len=81, off=7	O22: len=66, off=21
	O_5	O51: len=45, off =89	O52: len=34, off=87
F_{h2}	O_3	O31: len=19, off=0	O32: len=5, off=0
	O_6	O61: len=22, off=19	O62: len=18, off=5
F_{h3}	O_4	O41: len=101, off=0	O42: len=434, off=0
method		m1	m2

len: length off: offset

(b)



(c)

Figure 5.7 Example 1: Fragmented Class C. (a) A logical view of C with its physical fragments. (b) A logical view of C with its objects. (c) A disk view of allocated physical fragment files and method files

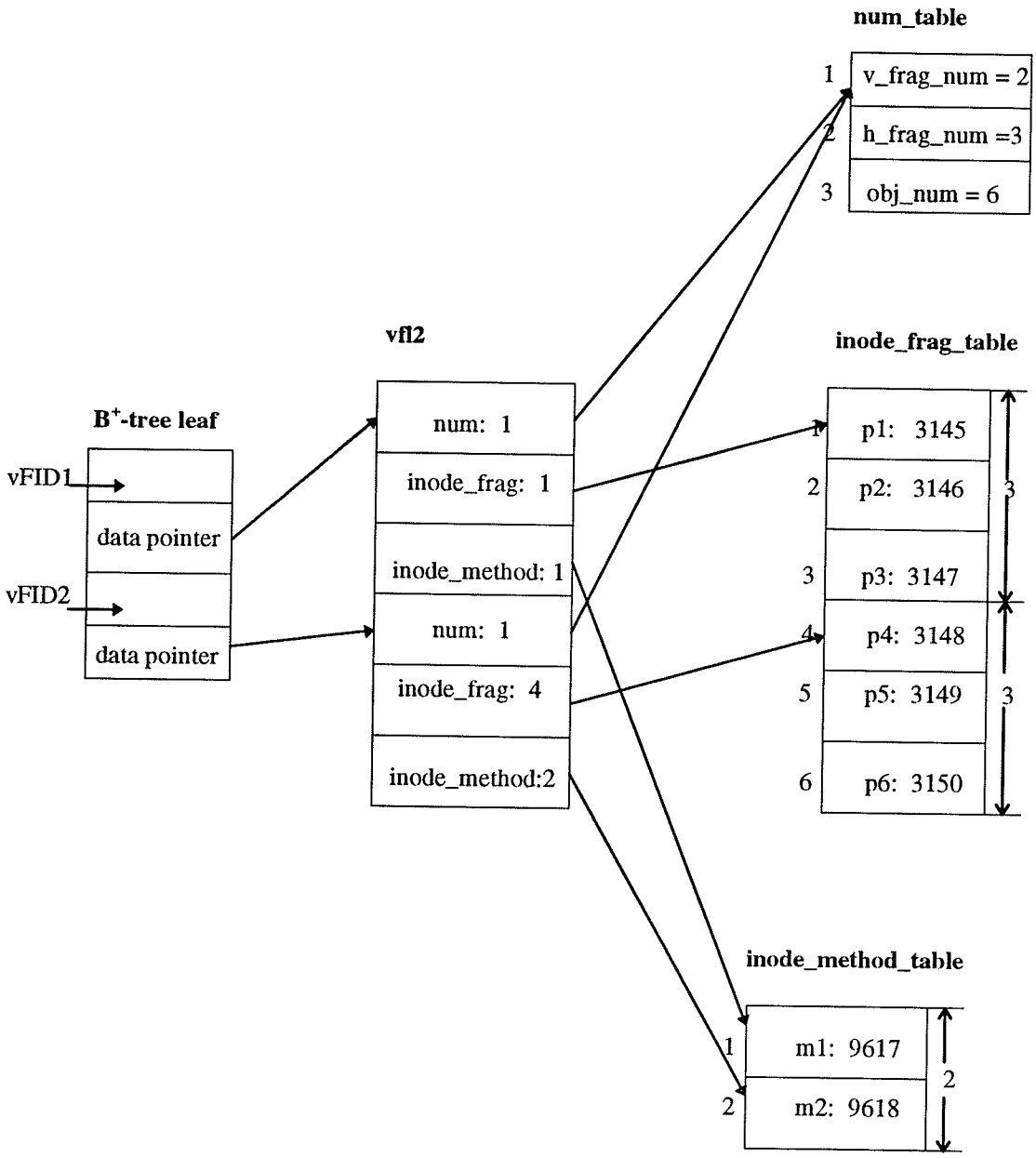


Figure 5.8 Vertical Fragment Index Tables of Example 1.

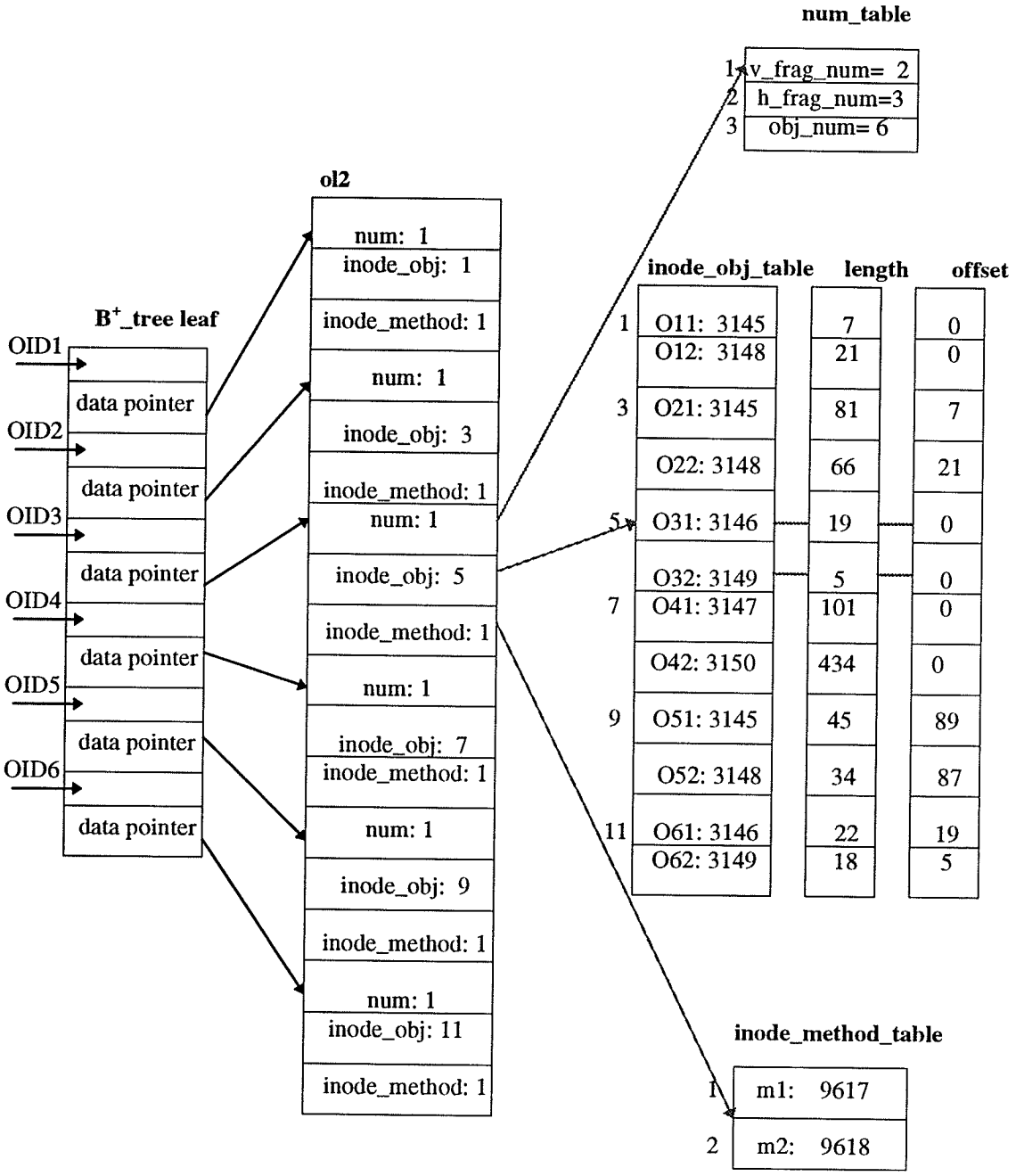


Figure 5.9 Object Index Tables of Example 1.

C_1: {O_1, O_2, O_3}

$F_{h1}=p_{11}$ O_1	length=26 offset=0
O_3	length=85 offset=26
$F_{h2}=p_{12}$ O_2	length=67 offset=0
method	m1

(a)

C_2: {O_4, O_5}

	$F_{v1}=p_{21}$	$F_{v2}=p_{22}$
O_4	length=99 offset=0	length=34 offset=0
O_5	length=81 offset=99	length=21 offset=34
method	m21	m22

(b)

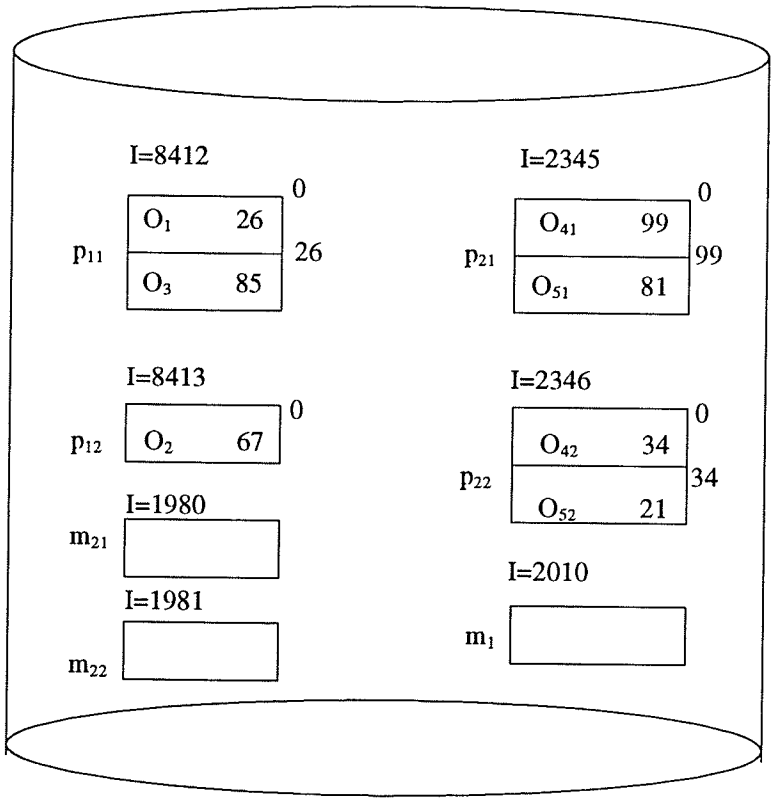


Figure 5.10 Example 2: Fragmented Classes C_1 and C_2. (a) A logical view of horizontally fragmented class C_1. (b) A logical view of vertically fragmented class C_2. (c) A disk view of the physical pragments.

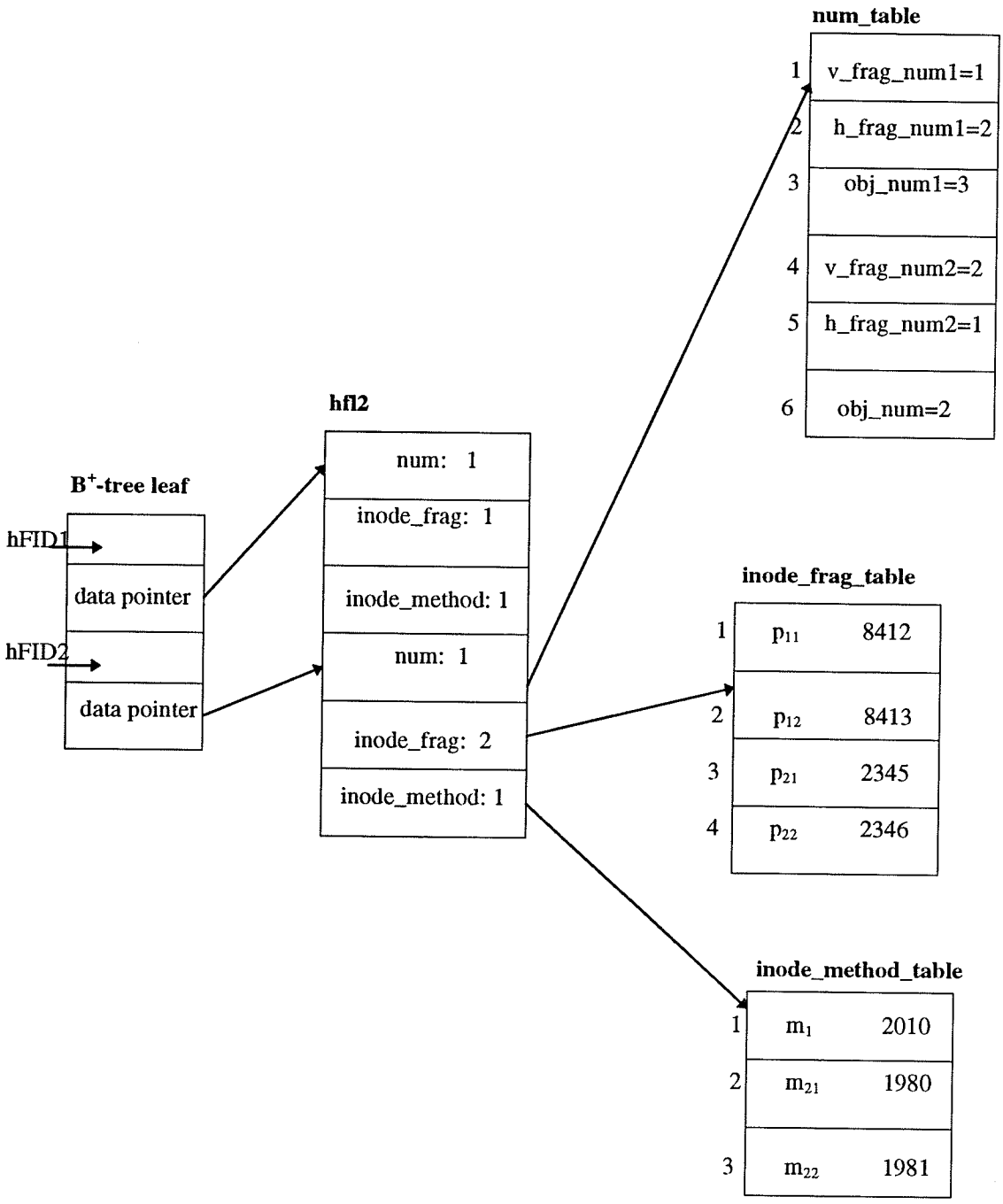


Figure 5.11 Horizontal Fragment Index Tables of Example 2

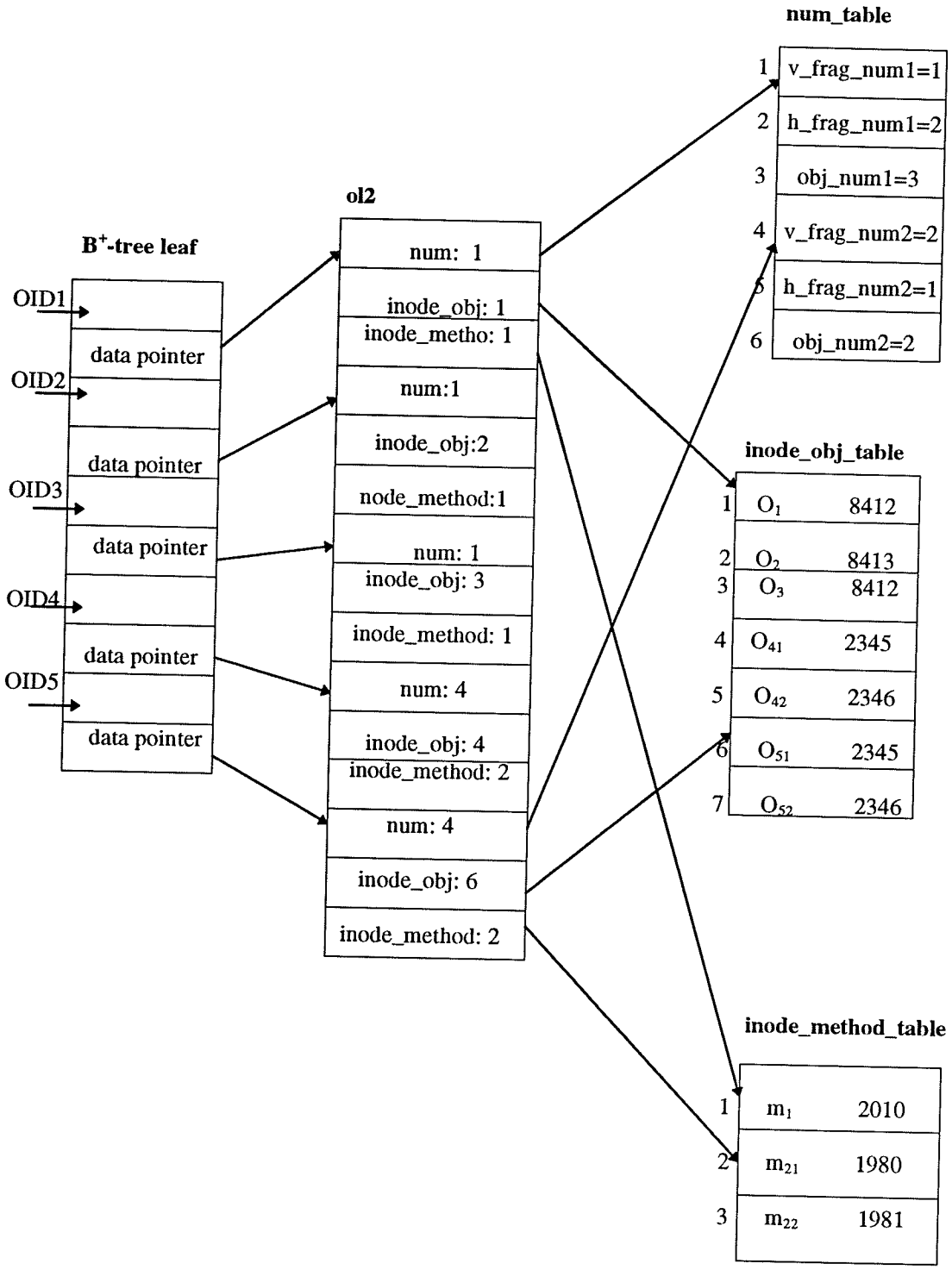


Figure 5.12 Object Index Tables of Example 2.

fragments. Class C_2 has 2 objects (O_3, O_4)and it is only vertically fragmented. C_2 has 2 vertical fragments (F_{v1}, F_{v2}), and thus it can only have 2 physical fragments (p_{21}, p_{22}) which are the same as the two vertical fragments. Figure 5.10 shows the logical and disk view of the two fragmented classes. Figure 5.11 gives the horizontal fragment index tables, and Figure 5.12 gives the object index tables of the two classes.

5.2.3 Data Search

Once the index structures are created, OSA can start satisfying data searches. The search process determines the inode numbers (and, in the case of object lookup, lengths and offsets corresponding to each inode) when given a fragment identifier or an object identifier.

Algorithm Search

```
if vertical fragment ID then
   Search vertical fragment B+-tree;
   Search vertical fragment second level index table;
   Search first level index tables inode_frag_table and inode_method_table;
   Return a set of inode-numbers for the requested vertical fragment;
end;
if horizontal fragment ID then
   Search horizontal fragment B+ tree;
   Search horizontal fragment second level index table;
   Search first level index tables inode_frag_table and inode_method_table;
   Return a set of inode-numbers for the requested horizontal fragment;
end;
if object ID then
   Search object B+-tree;
   Search object second level index table;
```

Search first level index tables inode_inode_table,inode_method_table, obj_length_table and obj_offset_table;

Return a set of inode-numbers of the object attributes and methods, a set of the object's lengths and offsets;

end.

Consider the processing required in searching for an object. Given an OID, OSA uses the OID as a key pointer to search in the B^+-tree to find the data pointer. Through this data pointer, OSA then determines the position of the relevant *num* in *ol2*. The next element, *inode_attr*, points to an index indicating the location of the first required inode number in *inode_obj_table*. Starting from this element, OSA takes *'v_frag_num'*, inode numbers from *inode_obj_table*. Using the same index, the corresponding data in *obj_length_table* and *obj_offset_table* will be obtained. The element following the *inode_attr* is *inode_method*, which points to the location of the first inode number of the methods belonging to this object. Starting from this inode number, OSA loads *'v_frag_num'* consecutive method inode numbers. At this point, all the inode numbers of the object and the relevant data (object lengths and offsets) have been obtained. This information may then be used to locate the bytes composing the object. Searching a logical fragment follows the rules shown in Figure 5.4 and Figure 5.5.

In Example 1 (see Figure 5.9), given an object identifier OID=3 (OID3), through the B^+-tree searching, the final key pointer will end up at certain location of the leaf level. The data pointer associated with this key pointer then points to the 7th entry of *ol2*. From this entry, three contiguous entries provide three pointers, *num, inode_obj*, and *inode_method*, which point to *num_table*, *inode_obj_table*, and *inode_method_table* respectively. Since the *inode_obj* pointer points to the 5th entry of *inode_obj_table*, the

physical fragment containing the first piece of the attributes of the requested object can be located. Starting from the 5th entry, contiguous *2* entries will determine the entire locations (inode number 3146 and 3149) of the physical fragments containing all the attributes of the requested object. The information 2 (*v_frag_num*) is provided by *num_table* which can be reached through the *num* pointer in *ol2*. Since objects' attributes are blended into physical fragments, accessing the object needs the lengths and offsets of the attributes in the provided physical fragment files with the inode numbers as 3146 and 3149. The length and offset data can be reached in *obj_length_table* and *obj_offset_table* using the same entry numbers in *inode_obj_table* where 3146 and 3149 reside. Thus, the attributes of the requested object (OID3) can be found at the physical fragment file (3146) with length=19 and offset=0, and at the other physical fragment file (3149) with length=5 and offset=0. The *inode_method* pointer of *ol2* points to the first entry in *inode_method_table* which contains the location of the methods. Starting from the first entry, contiguously *2* entries provide the files with the inode numbers as 9617 and 9618 where the methods of the requested object (OID3) reside. Again, the information 2 (*v_frag_num*) is from *num_table* through the *num* pointer.

In Example 2 (see Figure 5.11), given a horizontal fragment identifier hFID=2 (hFID2), the data pointer at the B^+-tree leaf level points to the 4th entry of *hfl2*. The next entry provides the *inode_frag* pointer which points to the 2nd entry of *inode_frag_table*. Through the *num* pointer at the 4th entry of *hfl2*, the information of *v_frag_num* =1 can be obtained at the 1st entry of *num_table*. Since only horizontal fragments exist (*v_frag_num* = 1), the physical fragment file (inode number = 8413) alone in the 2nd

entry of *inode_frag_table* provides the entire attributes of the requested horizontal fragment. For the same reason, the 1st entry of *inode_method_table* which is reached through *inode-method* pointer, provides the location (inode number = 2010) of the entire methods of the requested horizontal fragment.

Chapter 6

Distribution

To provide a very large data space, a persistent object store must eventually be distributed so that many sites may share the storage load. Distribution transparency provides a uniform mechanism for data operation both in local and remote nodes. This permits the distributed stores to be considered a single large storage system. This fits the concept of persistence transparency because all the physical properties, including placement of data, replication of data and the failure of nodes, are hidden from users.

A "bottom-up" approach may be used to extend the presented single-site object store system into a truly distributed *peer-to-peer* system, which, unlike a traditional *client-server* distributed system, does not have a single, centralized server. Every node on the network is capable of acting as a server to the other nodes, and the information stored in one node can be freely accessed by the other nodes. In such a system, information and resources can be easily shared among different nodes.

Distributed systems involve many complex and inter-related issues. In this chapter, only those issues relevant to extending the centralized object store into a distributed one are discussed.

6.1 Operating System Support

Most existing persistent systems have been developed above conventional architectures or operating systems which have little support for efficient object-oriented programming in a distributed environment. The described object store is constructed on top of Mach, a modern operating system which provides all the underlying mechanisms needed to support shared, persistent objects[Mil93]. The advanced programming features of Mach such as kernel threads, IPC, and especially its virtual memory management which allows local, single-copy sharing of code and data, object faulting and transparent on-demand object access, make the implementation extensible to a distributed system.

A memory manager under Mach, can serve as a Distributed Shared Memory (DSM) manager [Tev89, Bar90, Tan92, Boy93] which handles references to the shared pages of virtual memory. A shared page is either readable or writable. The readable shared pages may have a number of replicated copies on different machines, while the writable ones have only one copy. If a thread on a machine references a readable page, the DSM server will map a copy into the machine's memory for reading. If a writable page is referenced, the DSM server will request the page from the kernel on the machine holding the page and then map it into the referencing thread's memory. During these procedures, if the requested page is not in the memory, a page fault will be sent to the kernel. The kernel

then employs the external memory manager (the pager) to map the page from secondary storage.

Using the Mach operating system, a distributed shared virtual memory (DSVM) can be built which extends the concept of a single, large, shared, *persistent* virtual object space to tasks scattered across the different nodes in a distributed system [Mat95]. On each node, there is a physically independent object storage system to support the locally stored subset of the shared virtual memory. Together, all the object stores on the distributed nodes support a uniform DSVM system. With the same address space visible to all tasks on all nodes, this DSVM system can be viewed as a one-world model [Gra93] upon which user applications execute. A persistent object in an object store on any node is then regarded as a global persistent object and can be referenced at a pre-defined location in the DSVM by all nodes. Each object identifier provided by the object storage system can be uniquely mapped to a virtual memory address since [Gra93, Mat95]:

- virtual addresses within a shared virtual memory are system-wide and consistent, and
- the shared memory address space is never destroyed and is valid for all tasks across space and time.

Mach, running on 64-bit processors, can address 2^{64} bytes of virtual memory. This huge virtual address space, with the support of distributed object stores, provides users an ideal persistent object-oriented programming environment. Since each object uniquely corresponds to a virtual memory address, objects can be referenced in terms of their

virtual memory addresses thereby avoiding the overhead normally required to swizzle object references. When a user application from one node needs an object, it simply accesses the virtual memory address where the object resides in the DSVM. The DSM server running on Mach will perform the data mapping to the requesting node. If the requested object is not cached in a corresponding physical memory, a page fault will occur, and a pager will be called to page in the requested data from a local or a remote object store. The pager is, of course, also responsible for paging out the modified data to a local or remote object store.

Logically, each object resides in the virtual memory as a single independent unit with a unique virtual memory address as its object identifier. Physically, each object is blended in some fragments which reside in a certain object store. When an object is accessed through its virtual memory address, the system may choose to fetch the fragments involving that object into memory. This supports pre-fetching of related data since the fragments may contain data that the methods of the object need to access.

6.2 Distributed Fragment Allocation

In addition to supporting greater storage capacity, distribution is introduced to enhance overall system performance. There are two major factors that affect the performance of the applications in a distributed database system as described below ([Kar94]):

- Latency and bandwidth of disk I/O operations on each local site.

- Latency and bandwidth of data communication between sites.

In a distributed objectbase system, the above two issues can be dealt with as follows:

- Fragment the classes, and
- Allocate the resulting fragments on to various nodes appropriately.

The process of fragmentation determines the appropriate units of data for distribution. This can be done in an object-oriented environment by clustering related data in classes into groups based on class membership and analysis of data access patterns. Fragmentation is aimed at reducing the amount of irrelevant data accessed, and thus reducing the number of necessary disk I/O's. Distribution enables the effectiveness of class fragmentation. As discussed in Chapter 3, using class fragmentation as an object clustering scheme benefits a distributed system in many ways. This is the major reason for the object store to support fragments.

Fragment allocation [Kar94, Sen95] is concerned with the placement of the fragments onto the distributed nodes in such a way that the cost of communication among different nodes, the cost of accessing fragments, and the cost of updating fragments and their replicated copies on the other nodes are minimized. The problem of data allocation in a distributed environment has not been resolved so far. Karlapalem, et al. [Kar94] present some allocation algorithms which is the first work on fragment allocation in distributed object-oriented systems. The fragment allocation strategies required in a distributed object store system should follow the criteria described below:

- Allocation should place the fragments at the nodes from which they will be most frequently invoked. For fragments involving complex methods which invoke other methods, the invocation sequence should be considered in the allocation process.
- Fragments involving restricted access[1] and shared access[2] may have replicated copies at each of the nodes where they are frequently referenced.
- If the location of the methods is fixed, the fragments that involve those methods should be allocated at nodes based on the type of processing being done on the objects the methods access. If a complex object is accessed and then modified frequently, the fragments involving the component objects of that complex objects should be placed at the same node.
- If a complex object is accessed frequently from a node, the fragments involving the data belonging to that object may need to be allocated at that node.
- If inheritance is used, the fragments belonging to the super classes from which an instance variable inherits an attribute may need to be allocated at the same node with the fragment involving that instance variable.

6.3 Global Data Structures

After logical fragments are allocated to all the distributed nodes, building a local

---

[1] A set of objects O has the restricted access property if these objects are referenced by a restricted set of other objects R, via instance variables. Whenever the set of objects R is accessed, the set of objects O may also be accessed.

[2] A set if objects O has the shared access property if they can be referenced by *any* other object as an instance variable.

object store on each node may begin. The allocated logical fragments are first decomposed into physical fragments on a per node basis, and then the physical fragments are allocated to the local secondary storage. These procedures follow the same rules as those described in Chapter 4 and 5. However, building the *global* data structures for the object stores in a distributed system introduces some new considerations.

6.3.1 Global FIDs and OIDs

A persistent object in any object store of the distributed system must be persistent and consistent in the global environment. Thus, both the formats of an object identifier and a fragment identifier from the centralized object store must be extended with some distribution information. With respect to the conceptual object identifiers and the conceptual fragment identifiers, the new restriction is that the numbers associated with them should be not only unique to the local node, but also unique to the whole system. This means the conceptual identifiers for objects and fragments must be created uniformly and globally. One approach to this problem is to associate each conceptual object identifier with one and only one virtual memory address in the DSVM [Gra93], as was discussed earlier. This scheme also provides distribution transparency because such identifiers hide the physical locations of objects or fragments from users.

With respect to the internal object identifier and internal fragment identifier[3], some new information must be added. A field indicating the location of the node on which the object or the fragment resides is needed. The mappings required between conceptual

[3] Fragment identifiers are still required at the level of object store management.

identifiers and internal identifiers for both fragments and objects can be described by the following:

FID $\Rightarrow$ {#node, #controller, #disk, inode numbers of the physical fragments };

OID $\Rightarrow$ {#node, #controller, #disk, inode numbers of the physical fragments, the object segment lengths, and the segment offsets in the files}.

6.3.2 A Global Directory of Objects

As discussed in the previous chapters, each object store has an index system (Data Dictionary, see Chapter 4). Given an OID or a FID, using the index system, the store should be able to return a set of inode numbers and relevant information for locating an object or logical fragment. In the distributed system, the index systems of different object stores on all the nodes are united together to form a *global* index system, called the Global Directory of Objects (the GDO) [Mat95]. Given any global OID, or FID, the GDO will return the corresponding information for locating the requested object or fragment. Since the system is a peer-to-peer distributed system, any failure of a single node must not affect data manipulation among the other nodes. With the recovery functionality of the Data Dictionary as discussed in previous chapters, a local object store suffering a system failure should be able to be restored quickly. Constructed with care in such an environment, the GDO as a whole has certain reliability.

The GDO can be managed in different ways as described by Mathew et al. [Mat95]. Perhaps the simplest way is to keep the original index system belonging to an

object store on its own node. This scheme can be viewed as a natural partition of the global GDO across the distributed nodes. The reliability and the consistency of the GDO is maintained by the reliability and consistency provided by each Data Dictionary on the nodes. Although direct and easy to implement, this scheme has a serious disadvantage. If on one node, an object is requested which cannot be found through the local index system, then, in the worst case, the server of the node will have to inquire with all other local index systems on the other nodes to locate the object. This scheme can quickly cause excessive communication overhead.

Another option is to extend each local index system to reflect the whole GDO. That is, each node has a full, replicated copy of the GDO. This scheme enables data searching to be carried out locally. It greatly reduces the network traffic resulting from index services and speeds up the location of objects. However, this scheme may make it difficult to maintain the consistency of the GDO when insertion, deletion and migration of GDO entries are involved. Every update of the GDO will cause modifications on all of the nodes and induce related network traffic. In addition, this scheme increases the storage overhead needed to maintain index systems, because keeping a full GDO at every node consumes a great amount of storage in the system.

A balanced scheme between the above two is a partitioned GDO with limited replication [Mat95]. This scheme extends the local index system to a certain degree. On each node, the local object store possesses some portions of all index systems including those originally created in the other object stores. As a result, the distributed object stores can be organized into a number of groups in terms of their index systems. Each group has

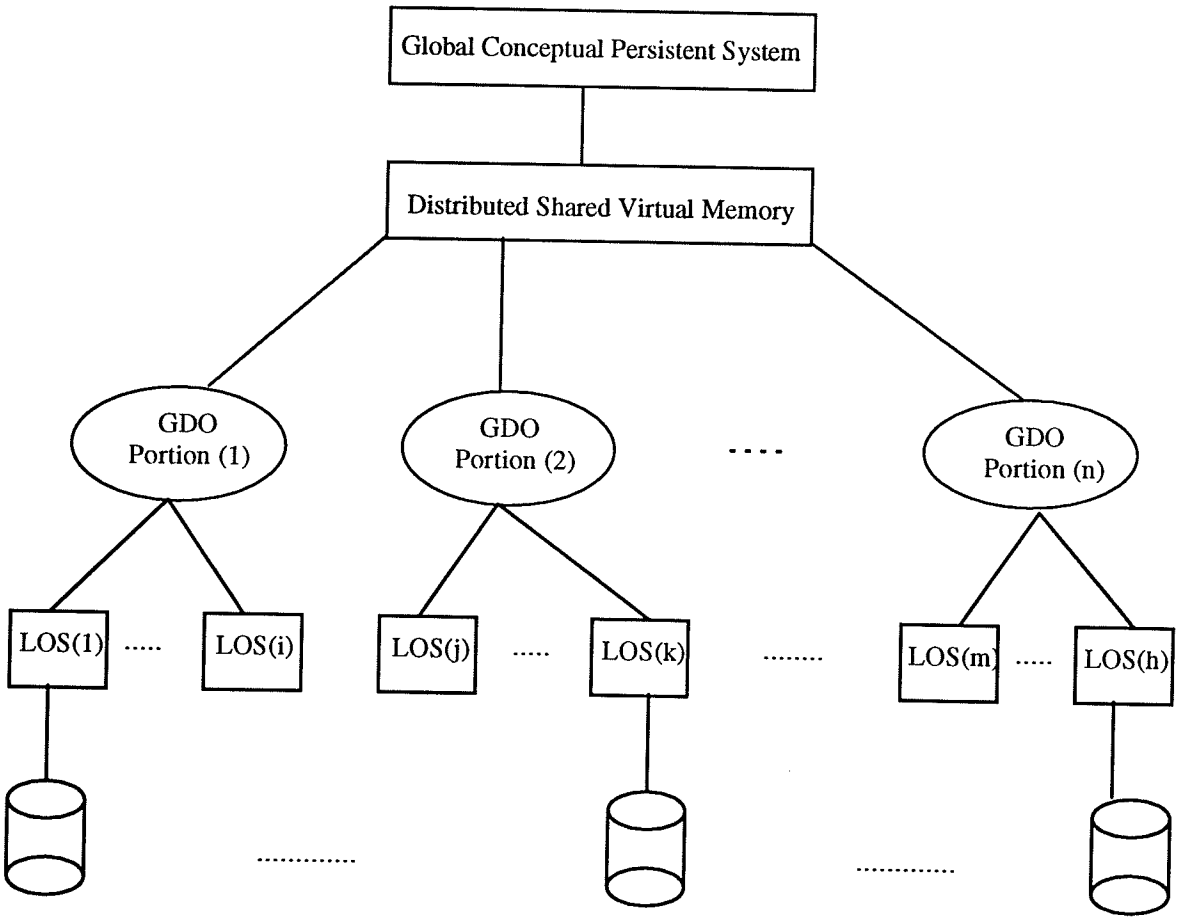
a totally different portion of the GDO, while inside each group, all the object stores have the same portion of the GDO which is an extended index system based on their locally created ones. This scheme keeps the advantages of the above two schemes and avoids the disadvantages to some extent. It is still relatively easy to implement and, when combined with efficient allocation strategies (e.g. allocating the related fragments in the same group where object stores have the same portion of the GDO), the performance of data retrieval can be greatly enhanced. Although the problem of the GDO consistency still exists, it can be limited by the use of a reasonable degree of replication.

As discussed in the previous chapters, the index structures of the object store are dynamically created when the system restores from a system failure or a shut-down operation. Therefore, each GDO portion in a distributed system is also created dynamically. The reasons that the dynamic scheme is used instead of storing the GDO portion on its local disk and loading it to the memory when the system starts include:

- The use of the B^+-tree, as the top level index structure of a GDO fits a dynamic scheme much better. It is complex to store the B^+-tree structure on disk.
- The dynamic scheme avoids the work of conventing the GDO disk format to its memory format which would otherwise increase the GDO creation time.
- The dynamic scheme greatly reduces disk I/O which is slower than memory access since the dynamic scheme only needs to load the first level index structure of the GDO from disk to memory.

- Because each node supports only a portion of the GDO, the B^+-tree and the other two levels of index tables should not be very large. Thus it will not take a long time to create the GDO on any given node.
- The dynamic scheme naturally supports the extension of the static object store system into a dynamic one in the future.

In summary, after fragments are allocated to each network node, the system will provide transparent access to physically distributed local object stores (LOSs). User applications are based on a Global Conceptual Persistent System which is defined on a Distributed Shared Virtual Memory (DSVM). Accessing objects in the DSVM is conceptually through a single version of the GDO which is physically partitioned to different groups of local object stores. In the same group, each object store possesses a copy of the same portion of the GDO which is different from that of the other groups. The system architecture for the GDO's implementation is shown in Figure 6.1.



LOS: local object store

Figure 6.1 Architecture of a Distributed Objectbase System system

Chapter 7

Conclusions

This Chapter will summarize the main contributions of the research work presented in the dissertation. Further, it will discuss possible future work in five directions:

- Implementation support for class inheritance.
- Clustering techniques.
- A dynamic distributed system.
- User transactions.
- Query systems.

7.1 Summary

The persistent storage of objects provides programmers with data manipulation capability at the level of programming languages without the need for explicit storage management. Object store support is essential in persistent object-oriented programming

and object-oriented database management systems. In this dissertation, a study of the basic concepts and requirements related to persistent object stores was undertaken. In particular, a detailed study of the design and implementation of an object store prototype was presented.

Supporting class fragmentation in the object store is a special feature which distinguishes the presented object store from other existing object stores. Object clustering, which determines the data storage format and physical storage units in the object store, is derived using the class fragmentation approach. Fragmentation promises to enhance application performance by reducing the amount of irrelevant data accessed and the amount of data transferred unnecessarily between distributed sites. Supporting fragmentation enables the object store to accommodate the object-oriented features of encapsulation, inheritance, and aggregation.

File systems are the most commonly used strategy to store and retrieve persistent data treated as a series of untyped bytes. Since object models have their own data semantics and data operations, new features should be added to a file system to develop an object store. The described prototype employs parts of the Berkeley Unix FFS and stores fragments/objects in a way which tries to keep related data as physically close together as possible. In addition, fragment/object index structures are constructed and function together with the FFS to provide "naming" services and to manage the object store.

The main features of the presented object store are the following:

- *Persistence*

The major function of the object store is to keep objects persistent and accessible. Persistence is achieved by using the FFS to store fragments and their access structures. Every object/fragment, together with its object/fragment identifier is persistent in the object store. Persistence enhances usage simplicity.

- *Correctness*

  Prototype testing shows that, given any object/fragment identifier, the object store always returns the corresponding object/fragment correctly. As such objects and logical fragments are reconstructable from their storage representations. No attempt at formal verification was made.

- *Design Simplicity*

  The strategy of using the FFS reduces the complexity of the design and implementation of the object store. Object store creation is straightforward and object/fragment lookup is easy to use.

- *Efficiency*

  Several points contribute to the efficiency of the object store. They are as follows:

  - Advanced architecture - Built directly on the advanced Mach microkernel, the object store will achieve better performance by avoiding unnecessary operating system overhead.
  - 64 bit addressing - Availability of 64-bit addressing eliminates the effort of certain object management functions which usually have to be considered when constructing an object store.

- Fragmentation approach - The fragmentation approach [Eze94a, Eze94b, Eze95] will result in improved performance in a distributed system.
- Fast data lookup - Using inode numbers, instead of the pathname-based naming system, to retrieve objects/fragments will decrease the overhead of file access. The simplicity of the index structures contributes to the efficiency of data lookup.
- Fast recovery - The strategy of keeping key index structures persistent allows the object store to recover quickly from software failures.

- *Flexibility*
  - Availability of two data interfaces - The object store provides an interface for accessing objects and an interface for accessing fragments. This strategy is designed to meet the requirements of different applications.
  - An adjustable file system - The FFS parameters may be adjusted based on empirical analysis and testing to find the best possible values for object storage for different application requirements.

- *Extensibility*

  The object store prototype provides a basis for a distributed object base system. The idea of using the Mach microkernel and fragmentation addresses a number of distributed system issues. The data structures of the centralized object store prototype can be extended to suit a distributed object base system. Avoiding sophisticated high level features enables this prototype to be further developed and

thus supports either object-oriented databases or persistent object-oriented programming languages. Finally, this prototype may be extended to support other forms of object grouping based on locality.

7.2 Future Work

Based on the successful centralized prototype of the object store, further research will aim at improving and developing it into a more practical system. Five possible directions for future work will be discussed in the following sections.

7.2.1 Implementation Support for Class Inheritance

Perhaps the most obvious future work is to extend the implementation to support class inheritance. To do this, in addition to class fragmentation information, the object store creation code requires the class hierarchy relationships as input. Each class should be assigned a record which specifies the class' position in the class hierarchy. The implementation will allocate the fragments belonging to a class to a subdirectory under its superclass' directory. In this way, the FFS layout policy will help to keep the objects with inheritance relationships physically as close as possible on the disk.

For the multiple inheritance case, the class inheritance hierarchy has to fit the filesystem tree hierarchy[1]. For this reason, the fragments of a class which has more than

[1] Note that there is no apparent FFS optimization for file links which might otherwise better support multiple inheritance.

one superclass, should be allocated to the subtree of one of the superclasses from which, compared with the other superclasses, the most information will be referenced.

7.2.2 Clustering Techniques

The clustering technique used in the object store is based on class fragmentation

[Eze95]. In spite of its advantages, it is not the only possible strategy for exploiting object semantics in clustering. Because of its important effect on performance, clustering techniques have been used by different object base systems [Ber94]. In the future, one consideration in clustering may be aggregation relationships [Bin91, Sch77]. This approach groups multiple hierarchic segments together and as a result, objects' sub-components can be stored immediately following them. This benefits queries that access an object and require navigation through its aggregation hierarchy [Ber94]. Another alternative is to cluster together larger objects as a "fragment" [Bil92, Oli94] if the object store frequently handles objects which have size greater than the page size. Still another consideration is to apply code profiling to an existing object store to determine clustering relations. The system then works out a clustering scheme which provide the minimum access cost, based on a statistical record of application access requests and access cost [19,20]. Finally, a hybrid clustering method may be adopted which takes the good points from different clustering techniques.

7.2.3 A Dynamic Distributed System

Currently the object store is created statically. This is because fragmentation information is generated statically. Although simpler, a static system is not flexible enough. For example, it can not handle any newly created objects. Thus once dynamic fragmentation is understood and practical algorithms have been developed, this object store should be extended to be more dynamic.

In the design of the prototype, some strategies appropriate for dynamic systems such as using the FFS and suitable data addressing approaches, were selected. The FFS is designed to support a dynamic file system. The FFS allows files to grow and shrink, which in turn allows objects/fragments to be created, modified, and deleted at any time. Since OIDs/FIDs are used as the indexes of object/fragment tables, objects/fragments can be addressed indirectly. This allows an object/fragment to be relocated without changing its unique logical address (OID/FID) in the object store.

7.2.4 User Transactions

To tolerate hardware and system failures, logging is widely used as a recovery method especially in a transaction processing system. In future, the object store may be re-implemented using local log-structured file system, and may log operations currently performed the FFS [Ros91]. There are two features of the log-structured file system that will make it desirable for transaction processing [Sel93]:

- A large number of dirty pages are written contiguously.

- The file system is updated in a "no-overwrite" fashion so that no separate log file is required.

Following Seltzer's work [Sel93], a transaction system can be embedded in the log-structured file system to support user-level transactions. Furthermore, in a distributed system a striped network file system such as Zebra [Har93] can be applied to the log-structured file systems. The Zebra architecture promises to provide cost-effective, scalable, highly-available network file system that can support high throughput file service. Combining striping and log-structuring may provide high performance, recoverable, distributed storage for object store implementations.

7.2.5 Query Systems

The object store can be used to support a persistent object-oriented programming system or an object-oriented database system. In the latter case, a query system is very important to provide sufficient functionality for users to retrieve the information in the object store. For a persistent object-oriented programming system, some query facilities may be embedded in the programming languages [Han93] though this is uncommon. For an object-oriented database system, a query interface using an SQL-like user language with formal semantics defined by object calculus and object algebra [Pet94] may be considered. An object store must provide arbitrarily complex search conditions and support efficient access to not only a single object, but also to sets of objects belonging to one or more classes to support any given queries. Class fragments implicitly support

efficient access to specific sets of objects. Incorporating explicit support for declarative queries is another area of future research.

Bibliography

[Agr89] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. *Proceedings of the ACM, SIGMOD Conference on the Management of Data,* 1989.

[And87] C. Andrews. Combining Languages and Database Advances in an Object-Oriented Development Environment. *Proceedings of 2nd International Conference on Object-oriented programming systems, languages, and applications*. pp. 430-40. 1987.

[Atk83] M. P. Atkinson, *et al.* An Approach to Persistent Programming. *The Computer Journal*, Vol. 26, No. 4, 1983, pp. 360-5.

[Atk94] M. Atkinson. Persistent Foundations for Scalable Multi-Paradigmal Systems. *Distributed Object Management.* M. T. Ozsu, *et al.* editors. Morgan Kaufmann Publisher. pp.26-50, 1994.

[Bal91] R. Balter et al. Architecture and implementation of Guide, an Object-Oriented Distributed System. *Computing Systems*, Vol.4, No.1, pp.31-67, 1991.

[Bar90] R.Baron, D. Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rashid, A. Tevanian and M. Young. Mach Kernel Interface Manucal. Technical Report, N00039-84-C-067. Carnegie-Mellon University, 1990.

[Bar92] D. Bartels and J. Robie. Persistent Objects and Object-Oriented Databases for C++. *C++ Report.* Vol.14, No.7, pp.49-50, 52-6, 1992.

[Ber92] S. Berman and K. J. MacGregor. A Persistent Class Store for Choices. *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pp.352-9, 1992.

[Ber94] E. Bertino, *et al.* Clustering Techniques in Object Bases: A Survey. *Data and Knowledge Engineering,* pp.255-75, Dec.1994.

[Bil92] A. Biliris. An Efficient Database Storage Structure for Large Dynamic Objects. *The Eighth International conference on Data Engineering,* pp.301-8, 1992.

[Bil93] A. Biliris al et. Making C++ Objects Persistent: The Hidden Pointers. *Software-Practice and Experience.* Vol.23, No.12, pp.1285-303, 1993.

[Bil94] A. Biliris and J. Orenstein. Object Storage Management Architecture. *Advances in Object-Oriented Database Systems.* A. Dogac al et. editors. Springer-Verlag, pp.185-200, 1994.

[Bin91] J. Bing et al. Effective Clustering of Complex Objects in OO Databases. *The Proceedings of ACM SIGMOD Conference,* pp,22-31, 1991.

[Boo91] G. Booch. Object-Oriented Design with Applications. Redwood City, CA. Benjamin Cummings. 1991.

[Boy93] J. Boykin, *et al.* Programming under Mach. Addison-Wesley Publishing Inc. 1993.

[Bre89] R. Bretl. The GemStone Data Management System. *Object-oriented Concepts, Applications, and Databases.* Addison-Wesley, 1989.

[Bro89] A. L. Brown. Persistent Object Store. Ph.D. Thesis, Universities of Glasgow and St. Andrews PPRR-71, Scotland, 1989.

[Bro92] A. L. Brown and R. Morrison. A Generic Persistent Object Store. *Software Engineering Journal* Vol.7, No.2, pp.161-8, 1992.

[Cam91] R. H. Campbell and P.W. Madany. Considerations of Persistence and Security in Choice, and Object-Oriented Operating System. Technical report. University of Illinois at Urbana-Champaign, UIUCDCS-R-91-1670. March 1991.

[Car90] M. Carey, *et al.* The EXODUS Extensible DBMS Project: An Overview. S. Zdonik and D. Maier, editors. *Readings in Object-Oriented Databases.* Morgan-Kaufmann, 1990.

[Cas93] M. Castro, N. Nevers, P. Trancoso, P. Sousa. MIKE: A Distributed Object-Oriented Programming Platform on Top of the Mach Microkernel. *Proceedings of the USENIX Mach III Symposium,* pp.253-72, 1993.

[Cha92] J. S. Chase, *et al.* Lightweight Shared Objects in a 64-bit Operating System. *Proceedings of OOPSLA'92,* pp.397-413, 1992.

[Cha94] J. S. Chase, *et al.* Sharing and Protection in a Single-Addressed-Space. *ACM Transaction on Computer Systems,* Vol.12, No.4, pp.271-307, 1994.

[Che92] Chevalier, Pierre-Yves. A Replicated Object Server for a Distributed Object-Oriented System. *11th Symposium on Reliable Distributed Systems*, pp.4-11, 1992.

[Che94] R. Chen, *et al.* Making C/sup ++/ a Persistent OOPL. *Information and Software Technology.* Vol.36, No.2, pp.119-25, 1994.

[Chr83] K. Christian. The UNIX Operating System. Wiley-Interscience Press, NY, USA, 1983.

[Das92] P. Dasgupta and R. J. LeBlanc. The Structure of the Clouds Distributed Operating System. IOS Press, Netherlands. pp. 36-60. 1992.

[Deu90] O. Deux, *et al.* The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering,* Vol.2, No.1, pp.91-108, March 1990.

[Eli90] J. Elliot B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, Vol.8, No. 2, pp.103-39, April 1990.

[Elm89] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, 1989.

[Eze94a] C. I. Ezeife and K. Barker. Horizontal Class Fragmentation in Distributed Object Based System. *Proceedings of the Second Biennial European Joint Conference on Engineering Systems Design and Analysis,* pp.225-35, 1994.

[Eze94b] C. I. Ezeife and K. Barker. Vertical Class Fragmentation in a Distributed Object Based System. *Proceedings of the Second International Symposium on Applied Corporate Computing, ISACC,* pp.43-52, 1994.

[Eze95] C. I. Ezeife and K. Barker. Comprehensive Approach to Horizontal Class Fragmentation in a Distributed Object Based System. *International Journal of Distributed and Parallel Databases,* Vol.1, No.1, 1995.

[Gla93] G. Glass. Unix for Programmers and Users. Prentice Hall Inc. Englewood Cliffs, NJ, 1993.

[Gra93] P. C. J. Graham, K. E. Barker, S. Bhar, and M. Zapp. A Paged Distributed Shared Virtual Memory System Supporting Persistent Objects. Personal Communication, Department of Computer Science, University of Manitoba.

[Gra94] P. C. J. Graham. Application of Static Analysis to Concurrency Control and Recovery in Objectbase Systems, Ph.D. Thesis, University of Manitoba, 1994.

[Haa90] L. M. Haas et al. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering,* Vol.2, No.1, pp.143-59.

[Hab90] S. Habert and L. Mosseri. COOL: Kernel Support for Object-Oriented Environments. *Proceedings of ECOOP/OOPSLA '90,* pp.269-77, 1990.

[Han93] E. N. Hanson, T. M. Harvey and M. A. Roth. Experiences in Database System Implementation Using a Persistent Programming Language. *Software-Practice and Experience.* Vol.23, No.12, pp.1361-77. December 1993.

[Har92] J. H. Hartman and J. K. Ousterhout. A Striped Network File System. *Proceedings of the USENIX of ECOOP/OOPSLA'90,* pp. 1-9, May 1992.

[How88] J. H. Howard, *et al.* Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems,* Vol.6, No.1, pp.55-81, 1988.

[Hud89] S. E. Hudson and R. King. Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM Transactions on Database Systems,* Vol.14, No.3, pp.291-321, 1989.

[Hur93] A.R. Hurson and Simin Pakzad. Object-Oriented Database Management Systems: Evolution and Performance Issues. *IEEE Computer.* Vol.26, No.2, pp.48-60, Feb. 1993.

[Joh93] T. Johnson and D. Shasha. The Performance of Current B-Tree Algorithms. *ACM Transactions on Database Systems*, Vol.18, No.1, pp.51-101, 1993.

[Kar94] K. Karlapalem, S. B. Navathe and M.M.A.Morgi. Issues in Distribution Design of Object-Oriented Databases. *Distributed Object Management,* M. Tamer Ozsu, U. Dayal and P. Valduriez editors. pp.148-64. Morgan Kaufmann Publisher, 1994.

[Ker81] B. W. Kernighan and J. R. Mashey. The Unix Programming Environment. *IEEE Computer,* Vol.14, No.4, pp.12-22, 1981.

[Kem93] A. Kemper and S. Kossmann. Adaptable Pointer Swizziling Strategies in Object Bases. *Proceedings of the International Conference on Data Engineering,* pp.155-62, 1993.

[Kho93] S. Khoshafian. Object-Oriented Databases. John Wiley & Sons Inc., 1993

[Kim90a] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, Vol.2, No.1, pp.109-24, March 1990.

[Kim90b] W. Kim. Introduction to object-oriented databases. The MIT Press, Cambridge, MA, 1990.

[Kim90c] W. Kim. Object-Oriented Databases: Definition and Direction. *IEEE Transactions on Knowledge and Data Engineering,* Vol.2, No.3, pp.327-41, 1990.

[Kor90] T. Korson and J. D. McGregor Object-Oriented Design. *Communication of the ACM*, Vol.3, No.9, 1990.

[Lef89] S. L. Leffler, and M. K. Mckusick, J.S. Quarterman. The Design and Implementation of the 4.3 BSD UNIX Operating System. Addison-Wesley Publishing Company Inc., 1989.

[Lev90] E. Levy and A. Silberschatz. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys*. Vol.22, No.4, pp.221-374, 1990.

[Lyo85] B. Lyon, *et al.* Overview of the Sun Network File System. Technical Report CMU-CS-84-137, Sun Microsystems Inc., 1985.

[Mad91] P. W. Madany, R. H. Campbell. Considerations of Persistence and Security in Choices, and Object-Oriented Operating System. Technical Report UIUCDCS-R-91-1670, University of Illinois at Urbana-Champaign, 1991.

[Mai86] D. Maier, *et al.* Development of an Object-Oriented DBMS. *Proceedings of First International Conference on Object-Oriented Concepts, Applications, and Databases,* pp.472-86, 1986.

[Mar83] S. T. March. Techniques for Structuring Database Records. *ACM Computing Surveys*, Vol.15, No.1, pp.45-79, 1983.

[Mat95] J. Mathew and P. C. J. Graham. Object Directory Design Issues for a Distributed Shared Virtual Memory System Supporting Persistent Objects. Technical Report 95-04, Computer Science Department, University of Manitoba, 1995.

[Mil93] B. R.Millard, P. Dasgupta, S. Rao and R. Kuramkote. Run-Time Support and Storage Management for Memory-Mapped Persistent Objects. *13th International Conference on Distributed Computing Systems*, pp.508, 1993.

[Mor90] R. Morrison and M. P. Atkinson. Persistent Languages and Architectures. *In Security and Persistence*, J. Tosenberg and J. L. Keedy editors, Springer-Verlag. pp. 9-28, 1990.

[Mul93] S. Mullender editors. Distributed systems. *ACM press*, 1993.

[Mun93] D. S. Munro. On the Integration of Concurrency, Distribution and Persistence. Ph.D. Thesis, University of St. Andrews, 1993.

[Nee82] R. M. Needham and A. J. Herbert. The Combridge Distributing Computing System. Addison Wesley, 1982.

[Nel88] M. Nelson, B. Welch, and J.K. Ousterhout. Caching in the Sprite Network File system. *ACM Transactions of Computer Systems*, Vol.6, No.1, pp.134-54, Feb. 1988.

[Ngu92] T. A. Nguyen, *et al.* PC++: and Object-Oriented Database System for C+++ Applications. *Proceedings of the Second International Symposium.* pp.109-15. Japan, April, 1991.

[Oli94] G. Oliver and A. Laurent. Object Grouping in EOS. *Distributed Object Management.* M. T. Ozsu al et. editors. Morgan Kaufmann, pp.117-31, 1994.

[Ous88] J. K. Ousterhout, *et al.* The Sprite Network Operating System. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society,* Vol.21, No.2, 1988.

[Ozs91] M. T. Ozsu and P. Valduriez. Principles of Distributed Database. Prentice-Hall Inc., 1991.

[Ozs94] M. T. Ozsu, *et al.* An Introduction to Distributed Object Management. *Distributed Object Management.* M. T. Ozsu al et., editors. Morgan Kaufmann Publisher. pp.1-24, 1994.

[Pet82] J. Peterson and A. Silberschatz. Operating System Concepts. Addison-Weley Press, 1982.

[Pet94] R. J. Peters. TIGUKAT: A Uniform Behavioral Objectbase Management System. Ph.D thesis. University of Alberta, 1994.

[Rit74] D. M. Ritchie and K. Thompson. The Unix Time Sharing System. *Communications of ACM,* Vol.19, No.7, pp.365-75, 1974.

[Ros91] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *Operating System Review.* Vol.25, No.5, pp.1-15. October, 1991.

[Ros92] J. Rosenberg, J. L. Keedy, and D. Abramson. Addressing Mechanisms for Large Virtual Memories. *Computer Journal*, Vol. 35, No. 4, pp. 369-75, 1992.

[Ros93] J. Rosenberg, A. Dearle. Keynote Discussion Session on Operating System Support for Persistence. *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pp. 54-60, 1993.

[Row87] L. Rowe and M. Stonebraker. The POSTGRES Data Model. Proceedings of the International Conference on Very Large Data Bases. pp. 83-95, Sept. 1987.

[Sat90] M. Satyanarayanan, *et al.* Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transaction on Computers,* Vol.39, No.4, pp.447-59, 1990.

[Sch77] M. Schkolnick. A Clustering Algorithm for Hierarchical Structures. *ACM TODS,* Vol.2, No.1, pp.27-44, 1977.

[Sel93] M. I. Seltzer. Transaction Support in a Log-Structured File System. *Proceedings of USENIX Computing Systems,* Vol.5 No.3, pp.305-35, Summer 1992.

[Sha91] M. Shapiro. Soul: An Object-Oriented OS Framework for Object Support. *Workshop on Operating Systems for the Nineties and Beyond,* July 1991.

[Sin93] V. Singhal, *et al.* Texaz: An Efficient, Portable Persistent Store. *Proceedings of the Fifth International Workshop on Persistent Object Systems.* pp.11-33. Pisa, Italy, September, 1992.

[Smi87] P. D. Smith and G. Barnes. *Files and Databases.* Addison-Wesley Publishing Company. 1987.

[Sou93] P. Sousa, and M. Sequeira, A. Zuquete, P. Ferreira, C. Lopes, J. Pereira, P. Guedes, J. A. Marpues. Distribution and Persistence in the IK Platform: Overview and Evaluation. *Computing Systems*, Vol.6, No.4, pp.391-424, 1993.

[Sub94] K. Subieta. Persistent Object Store for the LOQIS Programming System. *Microcomputer Applications.* Vol.13, No.2, pp. 50-61, 1994.

[Str93] H. Strickland. ODMG-93: The Object Database Standard for C++. *C++ Report.* Vol.5, No.8, pp.45-8, 1993.

[Tan92] A. S. Tanenbaum. Modern Operating Systems. Prentice Hall, 1992.

[Tev89] A. Tevanian and B. Smith. Mach: the Model for Future Unix. *Byte*, Vol.14, No.12, pp. 411-17. 1989.

[Tri92] A. Tripathi, *et al.* Management of Persistent Objects in the Nexus Distributed System. *Proceedings of the Second International Workshop on Object Orientation in Operating Systems.* France, September, 1992.

[Vau92] F. Vaughan, T. Lo Basso, A. Dearle, C. Marlin, and C. Barter.Casper: A Cached Architecture Supporting Persistence. *Computing System*, Vol.5, No.3, pp.337-63, 1992.

[Wal83] B. Walker, *et al.* The LOCUS Distributed Operating System. *ACM SIGOPS, Operating System Review,* Vol.17, No.5, pp.49-70, 1983.

[Wei88] D. Weinreb, *et al.* An Object-Oriented Database System to Support an Integrated Programming Environment. *IEEE Database Engineering* Bulletin, Vol.11, No.2, pp.33-43. 1988.

[Wil90] K. Wilkinson, P. Lyngbek and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, Vol.2, No.1, pp.63-75, 1990.

[Wil91] P. R. Wilson. Pointer Swizzling at Page Fault Time. *SIGARCH,* Vol.19, No.4, pp.6-13, 1991.

[Xia93] L. Xiao and R.H. Campbell. Object-Oriented Transactions in Choices. *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*. Vol. 2, pp.50-9. 1993.

[Yan94] F. Yang, *et al.* Study on the Store Technique of Persistent Object. *Acta Acustica.* Vol.19, No.5, pp.1-8,16, 1994.