MASTER IN COMPUTER SCIENCE

File Migration in a Distributed Environment

Bah, Souleymane

Award date:
1996

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année Académique 1995-1996

File Migration in a Distributed Environment

Souleymane BAH

Promotor : Jean RAMAEKERS

This master thesis is presented in order to obtain the title of Licencié et Maître en Informatique

Abstract

In this master thesis, we tackle the problem of the Hierarchical Storage Management (HSM) and we show how a proposed HSM solution can be integrated in the HSM-CL product.

HSMS-CL is a software product running on Unix systems. It allows to backup and archive Unix files through another product called HSMS running on BS2000 system.

HSM allows to migrate transparently an automatically less frequently used files from expensive and faster disks such as hard disks to less expensive slower with a huge amount of space such as optical disks, magnetic tapes, etc. After migration, files are still available online or nearline for applications or users.

We present two HSM products which are DataMgr and EpochMigration about their migration mechanisms.

After studying how a proposed HSM solution can be integrated in the HSMS-CL product, we present a prototype under a Sinix system (Sinix-L) which implements some migration operations and a way allowing to distinct migrated files, but also a way to intercept files access requests.

Résumé

Dans ce mémoire, nous abordons le problème de la gestion de mémoires hiérarchiques (HSM) et nous montrons comment le concept HSM peut être intégré dans le produit HSMS-CL.

HSMS-CL est un produit tournant sur des systèmes Unix. Il permet de sauvegarder et d'archiver des fichiers Unix grâce à un autre produit appelé HSMS tournant sur un système BS2000.

HSM permet de migrer de façon transparente et automatique les fichiers les moins fréquemment utilisés à partir de disques rapides chers tels que les disques durs vers des disques capacitaires, lents et moins chers tels que les disques optiques, les bandes magnétiques, etc. Après la migration, les fichiers sont encore disponibles online ou nearline pour les applications et utilisateurs.

Nous présentons deux produits HSM qui sont DataMgr et EpochMigration au sujet de leurs mécanismes de migration.

Après avoir étudié comment une solution HSM proposée peut être intégrée dans le produit HSMS-CL, nous présentons un proptotype sous un système Sinix (Sinix-L) qui implémente quelques opérations de migration et un moyen permettant de distinguer les fichiers migrés et les fichiers non migrés, mais aussi un moyen pour intercepter des requêtes d'accès aux fichiers.

Acknowledgements

This work has been done during my stay at SIEMENS-NIXDORF company at Rhines in Namur (BELGIUM).

It is with a great pleasure that I acknowledge all the persons who have allowed me to realize this work :

- Mr. Jean Ramaekers, computer science professor at Facultés Universitaires Notre-Dame de la Paix in Namur (BELGIUM), for accepting to direct this master thesis and for his helpful advice.

- The following members of SIEMENS-NIXDORF company for giving me the right environment and for their helpful suggestions : Mr. Nicos Piperakis, Mr. Benoit Hucq, Mr. Georges Rossigon and the members of their teams (RD34, RD35).

I dedicate this study to my parents.

Contents

Introduction

Hard disks are central components to the functioning and performance of any computer. Ideally, users could add the fastest and largest disks available to their systems.

Unfortunately, hard disks are expensive. But even if hard disks are cheaper, they are still susceptible to head crashes and other casualties for data and they need continuous backup.

Workstations and file servers running complex applications, such as those evolving digital imaging, video-on-demand, multimedia, etc. require huge amount of space. One page of text requiring four kilobytes of storage may grow to two megabytes with the addition of a single picture. A short multimedia presentation may need a gigabyte of storage while a small library of full motion videos could require one terabyte or more.

Although the price of disk storage is decreasing, continuously adding more disk storages to avoid out of disk space situations is not a cost effective solution. There are many hidden costs associated with disk storage that are not often initially considered. The storage administrators tasks which consist among others to install and configure disk drives, identify inactive files and move them to other storage media are a labour intensive storage management tasks. This implies a lost productivity of users being unable to use the system during hardware upgrades.

In most cases, users do not work with all the files and data to their hard disks at one time. In fact, many of the files are not used or infrequently used. So, it would be very interesting to monitor the "fill level" of storage.

Today's distributed environments require sophisticated storage management software that utilizes Hierarchical Storage Management (HSM) concepts. A software based on HSM concept move automatically and transparently infrequently used files from expensive and faster disk drives such as hard disks to less expensive and slower with huge amount of space storage such as optical disks, cartridges, tape libraries etc..

The purpose of this work is to show how HSM concepts can be integrated to an existing product called HSMS-CL. This product, running on a UNIX system, provides backup and archival files through another product called HSMS running on a BS2000 system.

This work is divided in six chapters.

Chapter 1 is a brief presentation of HSMS and HSMS-CL. In this one, we describe the main concepts used in chapter 5.

Chapter 2 is a detailed description of Hierarchical Storage Management (HSM) concepts.

Chapter 3 presents two existing HSM products regarding to their architecture and their migration mechanisms.

Chapter 4 starts by a brief description of UNIX filesystem (ufs) objects used in this chapter and the followings. Then, in this one, we will analyze different solutions of file migration.

Chapter 5 shows how HSM concepts can be integrated to the HSMS/HSMS-CL products.

Chapter 6 presents a file migration prototype proposal.

Chapter 1 Presentation of HSMS

Introduction

The Hierarchical Storage Management System (HSMS) is a BS2000 software product which supports data management on external storage devices in a BS2000 system [HSMS95]. It provides several useful functions as backup, archival and migration of BS2000 files.

To cope with the increasing part of the UNIX systems in the organization and to complete the offered services, a complementary software called HSMS-CLIENT [HSCL95] has been developed. This new product allows the UNIX administrators and users to backup and archive their files via HSMS.

HSMS uses the ARCHIVE product for the transfer of the manipulated files to or from the storage levels.

1.1. Concepts

As HSMS uses several concepts that don't exist in another products, a brief description of these ones is necessary for a good understanding of the following chapters. However, only the most important concepts will be explained in full details as they appear in the next chapters.

1.1.1. Hierarchical Storage

As HSMS is a hierarchical storage system, the storage space, so-called in the Hierarchical Storage Management (HSM) terminology the storage hierarchy, is organized in several levels. These levels have various characteristics regarding storage costs and access time behaviour.

In HSMS [HSMS94], the storage space is divided in three distinct levels :

1. Storage Level S0

The storage level S0 is the normal processing level. It is the one the users can access online. This level is managed directly by the file system Data Management System (DMS), for the BS2000 system and UNIX File System (UFS), for the UNIX system.

As it is the online level, it is the fastest and also the most expensive one. It is thus composed by magnetic disks with fast access time.

2. Storage Level S1

The storage level S1 is the nearline background level. The data stored on this level are managed by the HSMS program. It is composed by disks that are generally slower and larger than those ones used at the level S0. This storage level is reserved for saving BS2000 files.

3. Storage Level S2

The storage level S2 is the off-line background level. It consists of magnetic tapes or magnetic tapes cartridges. It is thus the slower and less expensive storage level of HSMS.

All the data at this level are managed by HSMS.

Storage level	User access	Media	Access	Access time
S0 processing level	direct	disks	online	very short
S1 background level	via HSMS	disks	nearline	short
S2 background level	via HSMS	optical disk magnetic tape or cartridge	nearline off-line	long

Figure 1.1 Storage Hierarchy in HSMS/BS2000.

1.1.2. HSMS Archive concepts

The archive is the basic HSMS management unit. HSMS [HSMS95] stores and manages all data saved by either backup, archival or migration in archives. HSMS distinguishes five types of archives, one archive for each of the basics functions it offers : backup, archival and migration in BS2000 system, and backup and archival in UNIX system. All the basic HSMS functions cannot be executed until the appropriate archive has been created.

Each HSMS archive consists to :

- the archive definition containing the archive's attributes,
- the associated archive repository,
- the save files containing the saved data.

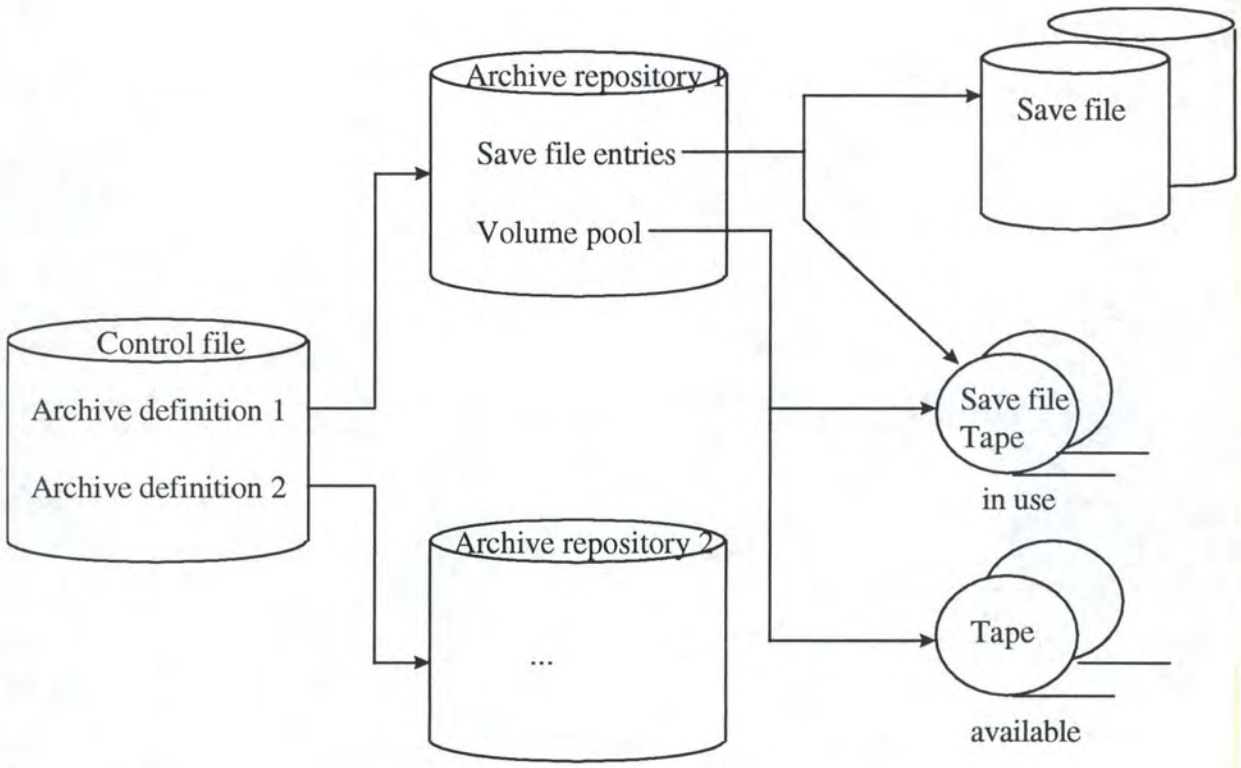


Figure 1.2 Structure of an HSMS archive.

1.1.1.1. Archive definition

The archive definition contains the following information :

- the type of the archive,
- the owner of the archive,
- the attributes of the archive (as the name of the associated repository and the accessible rules).

It is stored in the HSMS control file.

1.1.1.2. Archive repository

The archive repository contains the information about the saved data managed in the archive. These information are :

- the file names and properties,
- the save files attributes,
- the save versions attributes,
- the occupied and free save volumes.

The repository is managed by the ARCHIVE product.

1.1.1.3. Save file

The save file is a catalogued BS2000 disk or tape file. A save file on tape consists to a number of volumes having the same owner and the same Save File Identifier (SFID). A save file can contain one or more save versions.

1.1.1.4. Save version

A save version contains all files saved by a request plus the needed metadata to allow a restore or a recall of the files. A save version is identified by its Save Version Identifier (SVID). A file can be saved only once in a save version.

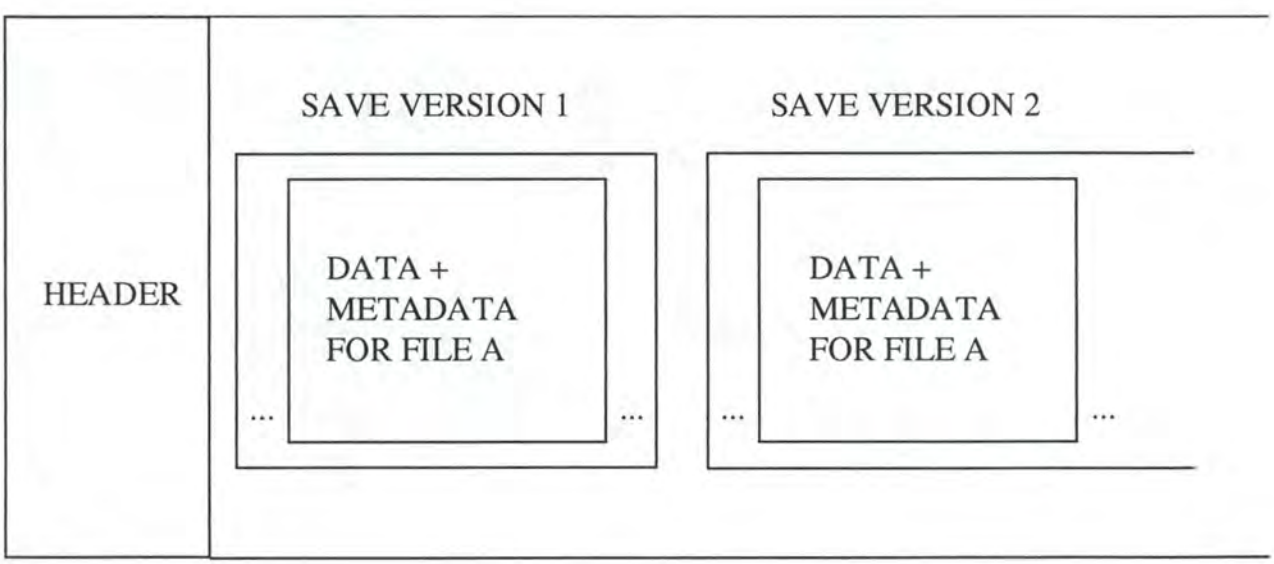


Figure 1.3 Sructure of a save file with several save versions.

A file can appear many times in a save file but only if different save versions are written in the save file.

1.2. Internal Structure

HSMS has access to user files only via the ARCHIVE product. It provides a number of server tasks for asychronous and parallel requests processing. The HSMS administrator defines this number of server tasks.

1.2.1. Task Concept

Server task/Main-task/subtask

HSMS and ARCHIVE use the task concept to process requests between their components.

1.2.1.1. Subtask/ARCHIVE-Main-task

The subtask concept has been introduced in order to achieve the parallelism of the data transport of the saved objects into the save files. The volumes containing the save files (to be continued or created) can be distributed among several devices. For each device, a subtask is created. One subtask is only dedicated to the data transfer between the saved objects and the saved files (in the two directions). The subtaks run independently from one another.

This mechanism is to shorten the saving and the reconstitution time by a parallel processing of tapes and/or disks during the archiving and reconstitution activities.

The subtasks are created by the ARCHIVE user task named ARCHIVE-Main-task or HSMS-Server-task. The ARCHIVE-Main-task divides the global processing in partial requests, so-called paquets, and transmit them to the subtasks for their processing.

1.2.1.2. HSMS-Server-task

The HSMS-Server-tasks are permanent system tasks that are created or deleted by the HSMS-Main-task. They process the user requests written in the request file. An HSMS-Server-task calls ARCHIVE and becomes an ARCHIVE-Main-task.

1.2.1.3. HSMS-Main-task

The HSMS-Main-task is a permanent system task. It realize the control of HSMS (initialization, creation or destruction of server-tasks, supervision and run of time-dependent request).

1.2.2. HSMS Files

HSMS uses the following files for synchronization and data transfer between tasks.

1.2.2.1. Control File

The control file contains the HSMS control parameters (number of server tasks to be created, device type for S1 level, definition of the archive, etc.). If the control file is inexistent at the start of HSMS, HSMS creates it with coded default values.

1.2.2.2. Request File

The request file contains the description of the requests to be processed by the server tasks. The request file is also used for the restart processing of interrupted requests. The request file is automatically created at start of HSMS if it was inexistent.

1.2.2.3. Archive Repository

A directory is defined by archive. It contains all needed information about files, tapes used and save versions or save files already created.

This file is maintained by the ARCHIVE product and HSMS has a read access to it.

In order to improve the access time to get information from the archive repository, a study has been done about this [DDK96].

1.2.2.4. ARCHIVE Result File

This file contains the report written by ARCHIVE. It is used by HSMS to produce its report. The result file is also needed for the restart processing.

1.2.2.5. Report File

A report file is used for all request that will produce a report. It contains the request protocol in internal form.

1.2.2.6. Print File

A print file is written and automatically printed/deleted for all report that must be sent to the printer. It contains the request protocol in printable form.

1.3. HSMS basic functions

In this section we present the basic functions of HSMS. These are :

- Migration,
- Backup,
- Long-term archival,
- Copying save files.

1.3.1. Migration

This function makes it possible to migrate user files from the processing level S0 to S1 or to S2 as well to recall these migrated files to S0 level for processing. This effectively reduces the danger of saturation of disk storage. If the disk saturation limit or the user allocations are exceeded, the migration function can be started and the inactive data migrated. The number of days since the last access, the minimum file size and the file fragmentation can be specified as criteria for selecting files to be migrated.

Files are automatically recalled before processing during file opening or reservation. Migration and recall can also be executed via instructions. The normal user has access to these statements.

Some files are normally excepted from migration (e.g. open files, temporary files or files that need to be repaired, HSMS files, etc.). The user can also set a migration lock.

When a file is migrated, the original catalogue entry remains on S0 and is given a 'migrated' symbol.

1.3.2. Backup

The concept of backup as a system service must be separated from the concept of long-term archival. The files on the processing level S0 and the migration levels S1 and S2 can be saved.

The system backup is carried out by the system or the HSMS administrator. HSMS manages backup data resources automatically and safely using system backup archives. For special applications, an application administrator can carry out the backup runs (for their own files) in their own backup archives.

Two backup facilities are offered : complete backup and differential saving (incremental backup). The principle of differential saving is to compare the catalogue entry information and the repository information and to save only files which have been changed since the last saving. Run time and memory space are then economized. Despite of that differential saving the possibility to reconstruct the complete data state as existing at the last saving always exists.

The backup function permits backup in S1 and S2 levels, either on disk, tape or cartridge. If operation is sometimes unmanned, the backup data can be temporarily stored on disk so that it can be transferred to tape at a later point in time. It is even possible to mix storage forms, i.e. to store the differential backup on disk and to store the complete backup on tape but this implies a new save file each time that we change the media.

Through the archive repositories, there are many options for selecting files for the purpose of saving back or reconstruction.

Via HSMS, it is also possible to make a complete backup by executing only an incremental backup (full from incremental). This function consists to run a incremental backup, then with the last differentials savings and the last full backup, to reconstruct a full backup on tape without needing to access to the objects.

1.3.3. Long-term archival

The HSMS archiving function, as an end user function, should first and foremost provides optimum support for long-term archival. It should also, as a basic function, implement management of all data on the archiving level. The HSMS archiving function provides the following additional services :

- Archive tapes are updated for independent archiving jobs (save versions) as standard. This means that tapes can be used in a more productive manner.
- A logical expiration date (specific to the job) can be allocated for archived files, independent of the tape's physical retention period. When changing archive tapes, those tapes that have not yet reached their expiration date can be specifically taken over.
- Archiving and reactivation job are collected in a job file. Tapes can be accessed at times specified by the system administrator. This enables the computer center to plan tape processing times better.

The data can be compressed when being transferred to the HSMS archive level and are automatically decompressed when accessed.

HSMS offers the option of adding extra user information with archiving jobs (names/descriptors and text for backup versions), in addition to improved archive information functions. Another option with HSMS is defining archives and setting characteristics such as maximum archiving period and default values for decompression, device types, etc.

1.3.4. Copying save files

HSMS allows backup files and the backup versions they contain to be copied within an archive or from one archive to another.
Copying can be used to :

- Swap backup files from S1 to S2,
- Copy backup files as a precaution against data loss,
- Change data volumes within long-term archives (recycle volumes),
- Reorganize a migration archive.

1.4. HSMS-Client

HSMS-Client is a software running on UNIX systems. It allows to backup and to archive UNIX files through HSMS on the BS2000 system. In order to do such operations, it requires the presence of Network File System (NFS) in server mode on the workstations and in client mode on the BS2000 system.

Moreover, the workstations file system must be mounted in the HSMS directory on the BS2000-UFS.

With this requirements fulfilled, and with the presence of the HSMS server software on the BS2000, it is possible to backup and archive the workstations files (node files) onto the BS2000 system.

The HSMS-Client software comes under three distinct forms : a Graphical User Interface (GUI), a Command Line Interface (CLI) and an Application Programmer Interface (API).

Chapter 2 Hierarchical Storage Management

Introduction

In this chapter, we will see the basic concepts of Hierarchical Storage Management (HSM). These concepts will allow us to understand in details HSM.

2.1. File Migration : What ?

File migration is concerned with the movement of less active files (i.e. files that are not regularly used) to slower, less expensive storage and leaving more active files on faster storage devices [HEU93]. But the file descriptor is retained on local disk to give transparent access to files. This file descriptor, often called stub file is used as a place holder.

File migration runs under a technology concept called **HSM.**

2.1.2. Hierarchical Storage Management Concept

The concept behind HSM is simple. Infrequently accessed data are automatically and transparently moved from more expensive, higher performance storage devices (hard disk drives) to less expensive, slower storage devices (tape drives and libraries, optical libraries).

When data is accessed it is automatically and transparently retrieved. HSM creates the illusion of infinite disk storage while maintaining high data availability. In theory, users never run out of disk space and have constant access to their data regardless of where they are stored.

HSM gives possible benefits including :

- Automated data management,
- Saturation avoidance,
- Decreased physical storage costs,
- Reduced storage management costs,
- Increased end user productivity,
- Optimal usage of storage devices.

2.2. Requirements

To be efficient, a file migration solution must satisfy the following requirements :

- **Automatization** File migration solution must reduce the costs of manually managing local and distributed storage.

- **Transparency** End users must not be exposed to the fact of their files are migrated physically on the storage server. In other words migration must allow migrated files to be accessed as if they were still on the client file system.

- **Reliability** The migrated data residing on the storage server must be protected against lost and/or corruption.

- **Portability** A file migration solution must be portable to a heterogeneous set of platforms. Example of platforms : SINIX, SVR4, Solaris 2.x, HP-UX.

- **Security** Confidential files through the network must be protected. Important files (e.g. bootstrap file) and HSM files must be excluded for migration.

- **Efficiency** Efficiency of a migration and recall mechanisms is the ratio of achieved performance (e.g. number of bytes migrated per seconds). Performance of migration and recall depends on several factors including the type of network, the size of files and the location of migrated files (e.g. it takes longer to recall a file from tape than optical disk). Migration and recall mechanisms must neither degrade the performance of the network nor the performance of the server and the workstation being managed.

- **Flexibility** A file migration solution must allow users to have various internal migration policies to determine *which* files get migrated *when* and *where.*

- **Standard** A file migration solution must be designed for open systems and industry standards. So, it must be related to the IEEE Mass Storage Reference Model. This model is presented in section 2.9.

2.3. File States

There are three types of file states : resident , non-resident and shadowed files.

Resident A resident file is a file that have not been migrated, both its attributes and its contents are located on the processing level (local disk). When first created, all files are resident files.

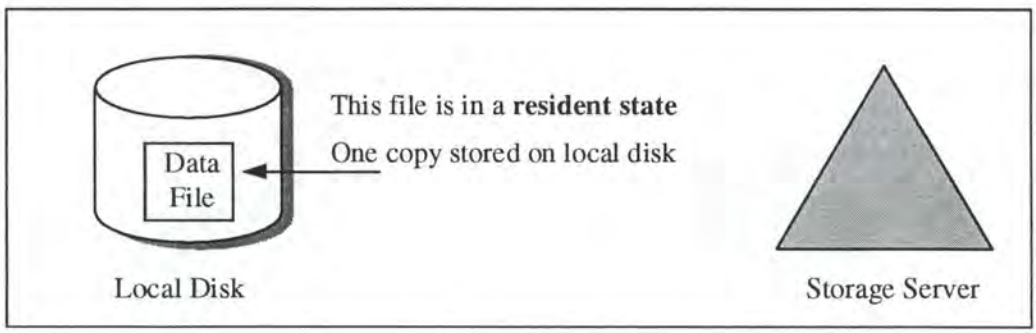


Figure 2.1 A file in a resident state.

Non-resident A non-resident file is a file that has been migrated. All the contents of a non-resident file are on storage server. A non-resident file is also called a stub file.

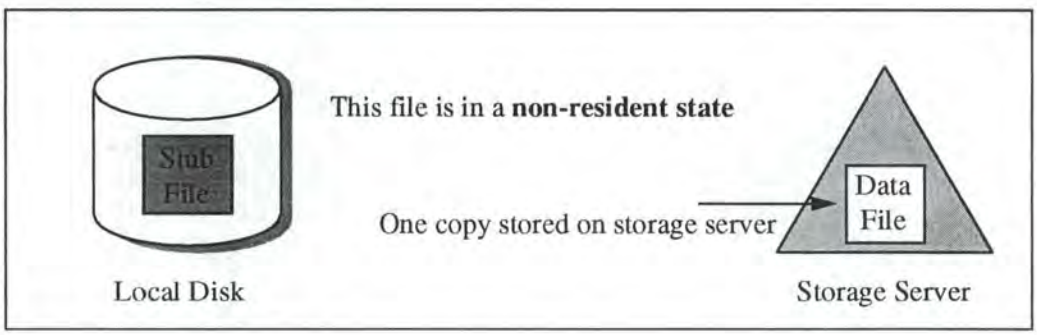


Figure 2.2 A file in a non-resident state.

Shadowed A shadowed file is a resident file for which a copy also exists on the storage server. A file becomes shadowed if and only if it is recalled or prestaged. If a shadowed file is modified, it enters in a resident state.
Prestaging is the copying of a file to the storage server without making it non-resident. The purpose of prestaging is to make subsequent migration faster.

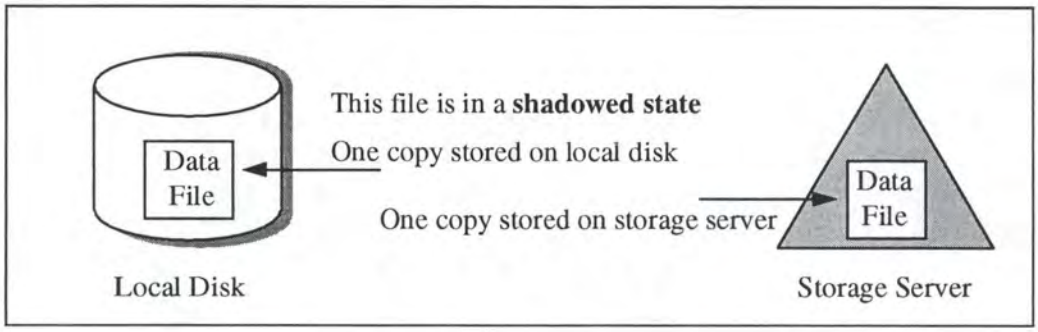


Figure 2.3 A file in a shadowed state.

2.4. Types of Migration and Recall

Migration

There are three types of migration : Explicit, Periodic, or Demand.

Explicit

Explicit migration is initiated by user request via a command.

Periodic

Periodic migration is initiated by an administrator job, usually via a cron, during non-peak times to bring file system utilization percentage down to a predefined threshold.

Demand

Demand migration is initiated by a daemon during normal operation, to make more space available on the file system when utilization reaches a predefined threshold. Thus, when this happens, user application merely pause until space is available instead of aborting with a "disk full" message.

Recall

There are two types of recall : Explicit, or Implicit.

Explicit

Explicit recall is initiated by a user request via a command.

Implicit

Implicit recall is concerned with the retrieval of migrated files. When a non-resident file is accessed (e.g. read), data are automatically transferred from the storage server to local disk.

For the explicit or implicit recall operation, a non-resident file is reloaded once to the local disk.

2.5. Migration Policies

Migration policies are concerned with the **Which**, the **Where** and the **When**. All the factors related to the migration policy are specified in a configuration file by the system administrator.

The Which

The *Which* is the answer to the question : which files are the best candidates for migration ? More often, these files are the least frequently used. The factors related to the which are described below.

- Minimum age,
- Minimum size,
- Files to be excluded from migration.

Minimum age

Minimum age specifies a threshold at or below which files are not be migrated.
The minimum age is the minimum number of days since the file was last referenced.

Minimum size

Minimum size is the minimum number of 1K blocks.

Files to be excluded from migration

Files to be excluded from migration are individual files and/or directories and/or particular files (e.g. bootstrap file, confidential files) that should be excluded from migration.

The Where

The *Where* is the answer to question : where a kind of files (e.g. big size files, old files, etc.) are migrated ? On tape or on optical disk, etc. The factor related to the where is :

- Storage level.

The When

The *When* is the answer to the question : when to start or to activate automatic migration processing. The when is related more often to the file system utilization percentage (watermark) and/or during non-peak times in order to increase free space on local disk. The factors related to the when are described below.

- Watermarks (high, low, prestaged).

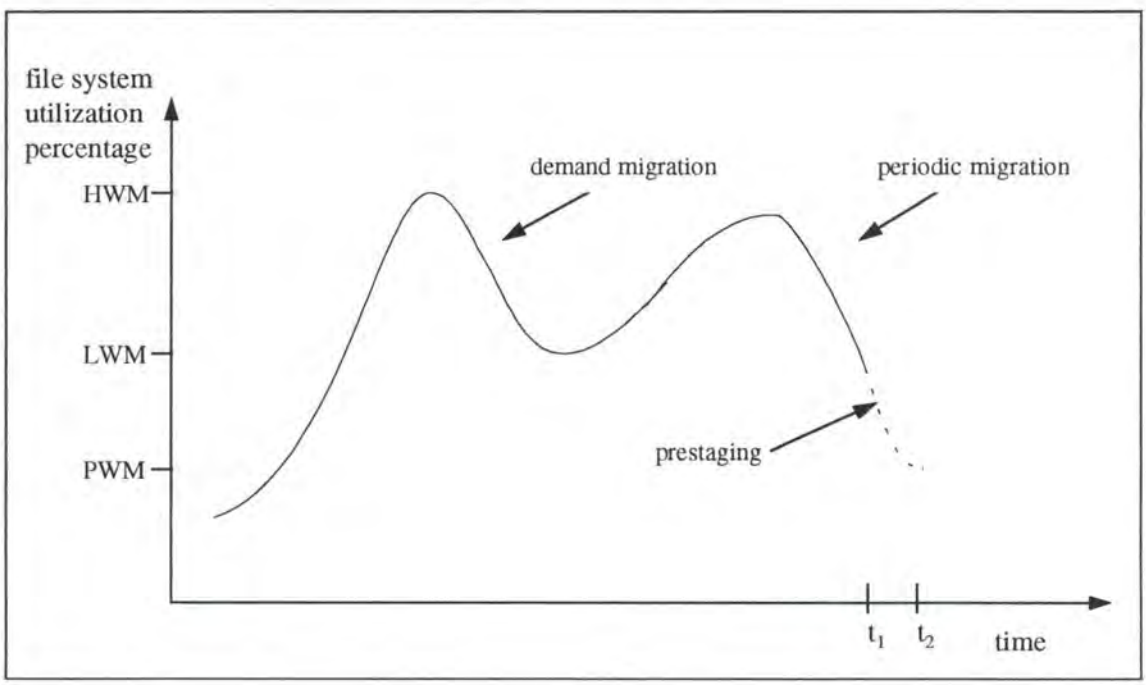
Watermark is a filesystem's preconfigured capacity threshold. We have three types of watermark

- **Low Watermark (LWM)** specifies the file system utilization percentage at which automatic (periodic or demand) migration will stop migrating files. If prestaging is in effect, LWM specifies when prestaging files.

- **High Watermark (HWM)** specifies the file system utilization percentage that the file system is allowed before starting the demand migration to make more space available.

- **Prestage Watermark (PWM)** specifies the file system utilization percentage at which periodic migration will stop prestaging files.
  Prestage watermark is always less than low watermark.

Example :



In this example, after periodic migration at time t_1 and assuming prestaging is in effect, prestaging can start from the LWM. At time t_2 the real file system utilization percentage is the low watermark.

2.6. HSM Storage Hierarchy

Storage Hierarchy

Storage Hierarchy is an assignment of storage units to different storage levels, depending on their availability, access time, and storage costs.

Cost increases and access time decreases going up the « storage pyramid » (see figure 2.4) while capacity memory increases toward the bottom of the pyramid [MIL91].

The bottom of the pyramid which usually consists of tape and optical disk have slow access time, on the order of seconds or minutes, and very lower costs. CPU cache or perhaps even CPU registers are on the top of the pyramid. They are more expensive, the smallest, the fastest in the hierarchy.

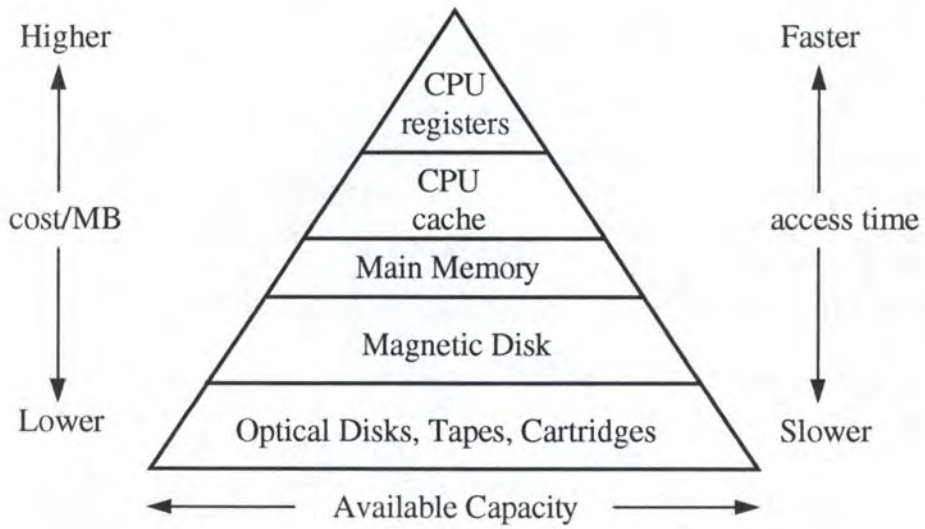


Figure 2.4 Storage hierarchy in Large Systems.

Storage Level

Storage level is related to a type of storage (e.g. tape, optical, disk) in the hierarchy where migrated files reside. A storage level depends on time access needed for the recall processing.

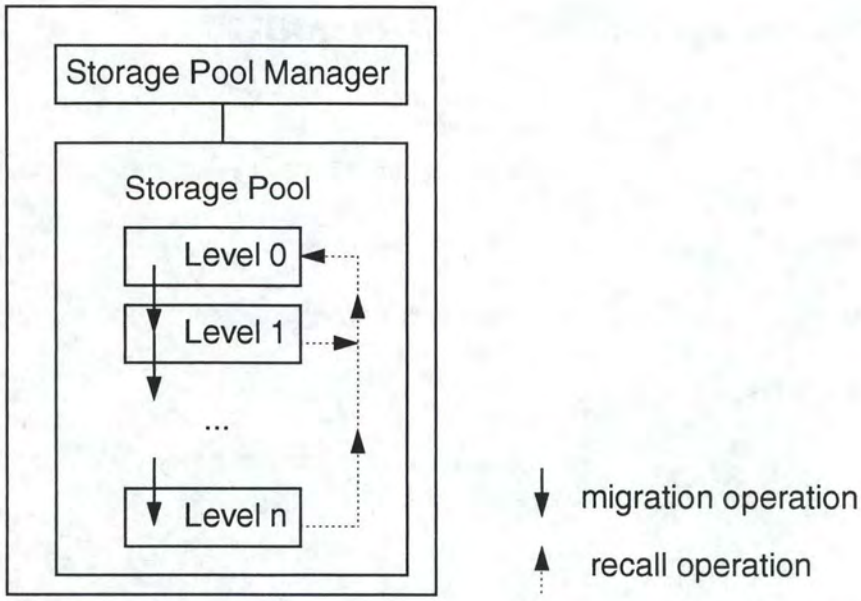


Figure 2.5 HSM Storage Levels.

Let N to be the number of storage levels, and $M_{ij}(f_i)$ the function which migrates the file f_i from the storage level i to the storage level j conforms to the HSM concept and $R_{ij}(f_j)$ recalls the file f_j from the storage level j to storage level i conforms the HSM concept.

Migration on Storage Levels $\forall$ i, $0 \leq i < N-1$, $\forall$ j, $0 < j \leq N-1$, $\forall$ f_i a candidate file on storage level i for migration, if $i < j$ then $M_{ij}(f_i)$.

Recall on Storage levels $\forall$ j, $0 < j \leq N-1$, f_j a migrated file and is a candidate for the recall from storage level j, $R_{0j}(f_j)$.

Example : Storage Hierarchy in HSMS/BS2000

In HSMS, the storage hierarchy (see figure 2.6) is divided in three levels : S0, S1 and S2.

Storage level S0

The storage level S0 is the normal processing level. It contains all resident files.

Storage level S1

The storage level S1 is a background level implemented by disk storage at lower cost. This storage level is reserved to the saving of HSMS files.

Storage level S2

The background level S2 consists on archived, backed up and migrated files on optical disk, magnetic tape or magnetic tape cartridge.

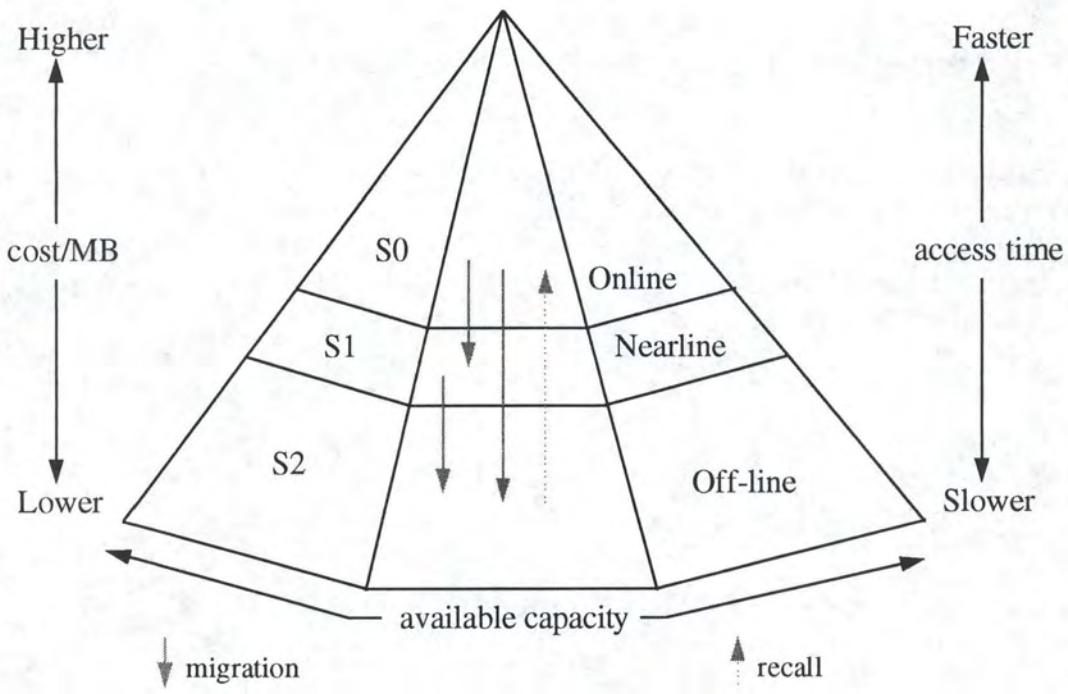


Figure 2.6 Storage Hierarchy in HSMS/BS2000.

2.7. HSM Levels

As in most product categories, comparing HSM systems, HSM vendors offer different array of features. Therefore it is useful to classify the various HSM systems into "HSM levels" according to their features [LIN94], [STE95].

Do not confuse *storage level* and *HSM level*. A storage level is related to a storage where a type of files (e.g. non-resident) resides, whereas HSM level is a category of HSM systems which have the same characteristics.

There are 5 HSM levels that are widely accepted as guidelines by the HSM industry.

Level 1 is a simple automatic file migration with transparent retrieval. By definition, all HSM products must meet level 1 requirement.
Example of level 1 HSM vendors :
- Advanced Software Concepts (Netarchive).
- Nestor Inc. (Nestor HSM).

Level 2 improves on level 1 by providing multiple watermarks or predefined thresholds, allowing administrators to tune an HSM system to meet particular needs of an organisation. Level 2 can manage two or more layers (levels) of nearline storage (e.g. optical jukebox, magnetic tape library). Level 2 products are for companies that have applications whose disk requirements vary greatly.
Example of level 2 vendors :
- Computer Associates International Inc. (Unicenter).
- Digital Equipement Corp. (Polycenter).

Level 3 provides transparent of three or more layers (levels) of storage hierarchy. Product of this category perform *volume management, media management, media optimization, and job queuing*. Level 3 product are for companies that have multiple layers of hierarchical storage (hard disks, optical jukebox, magnetic tape library).
Example of level 3 HSM vendors :
- Corner Storage System Group (Corner HSM).
- Lachman Technology Inc. (Open Storage Manager-Naperville, III).

Level 4 provides classification of files for migration. Migration is based on type and other criteria through the use of policies. For example, an administrator can classify each file according to type, size, location, or ownership. Next, the administrator can set different migration policies for each classification. A large number of HSM systems, especially those for UNIX, offer level 4.
Example of level 4 HSM vendors :
- OpenVision Technologies Inc. (HSM Extention).
- Alphatronix (Emissary/HSM).

Level 5 HSM level 5 adds object management. Objects are treated as structured or non structured records, or as nonfile structures. Level 5 identifies products that can work with database manager software such as Oracle, NT server, DB2/2, to migrate a segment of a database (rather than the entire file) to and from secondary storage.

2.8. Integrating backup and archival

Migration is not the replacement of data backup and archival. The purpose of backup and archival is to enable recovery of lost or inaccessible data and retrieval of point-in-time stored data. Whereas the purpose of HSM is to free space on higher performance, higher cost storage.

A storage management solution should have the ability, if desired by the end user, to verify that backup copies of data exist before data can be migrated. In addition, it should manage the integrity of data stored on the storage server.

In general, there are advantages to integration. Administrators can be assumed that the migration library is backed up; they can work with a single interface for both HSM, backup and archival and they can share devices for the three functions.

2.9. IEEE Mass Storage Reference Model

In this section we present a summary of the IEEE Mass Storage Reference Model version 4 [IEEE90].

The IEEE Mass Storage Reference Model was developed by the IEEE Technical Committee on Mass Storage System and Technology. This model defines how Mass Storage Systems should be implemented to response the need for standardised method of constructing memory storage systems.

The purpose of this model is to encourage the vendors to develop interoperable components that can be combined to form integrated storage systems and services.

In the same way as ISO seven-layered communication model, the IEEE Mass Reference Model provides a consistent set of concepts and terminology.

This model sets the foundation for the development of standard mass storage architectures and interfaces.

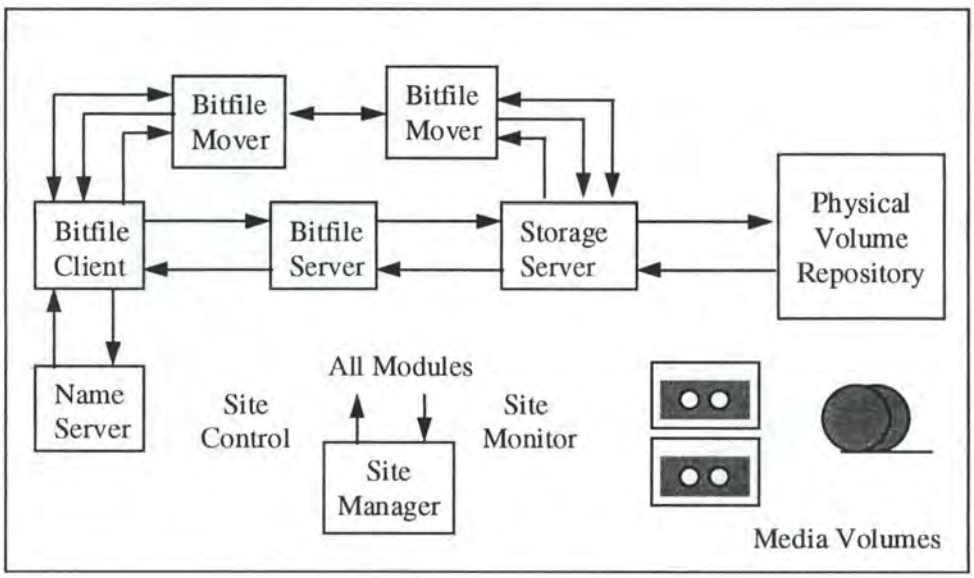


Figure 2.7 IEEE Mass Storage Reference Model.

The reference model partitions a mass storage environment into a set of related logical storage services (see figure 2.8).

Bitfile : is the content (uninterpreted data) of a non-resident file on the storage server. A bitfile has a unique identifier called bitfile ID.

The *Bitfile Client* presents an application-oriented storage abstraction.
The Bitfile Client defines concepts such as files, tables, blobs (Binary Large OBjectS), directory, file attributes, and access control.

The *Bitfile Server* provides the logical storage needed to implement the Bitfile Client.
The Bitfile Server manages the bitfiles.

The *Name Server* provides the mapping between the application oriented attributes (such as file names) and the IDs of bitfiles used to hold the file's data.

The *Storage Server* provides a set of logical volumes, which are the storage containers used by the Bitfile Server to hold bitfiles. These logical volumes may be associated properties such as size and location.

The *Physical Volume Repository* manages the real physical media used to implement the logical volumes. Its tasks include physical volume identification, access control, library unit control, and physical device access.

The *Site Manager* provides tools for monitoring and controlling the actions of the other services.

Chapter 3 Product Analysis

Introduction

In the previous chapter, we have seen the basic concepts of HSM. In this one, we will see how migration and recall operations are realized through two products which are DataMgr and EpochMigration. Why these products ? Mainly because migration and recall operations are based on different architectures but also because these two products are well documented.

3.1. Overview of DataMgr and EpochMigration Products

In this section, we present the DataMgr and EpochMigration products. We will see in addition, in one hand, the migration and recall functionality and in another hand, the migration policies provided by each product. But first, we will see the basic requirements satisfied by the two products.

3.1.1. DataMgr

DataMgr (read Data Manager) is one of the components of the AMASS Storage Management System (AMASS SMS) product family [DTM95]. This software product (DataMgr) in conjunction with an AMASS server provides a file migration service in distributed environments.

AMASS (Archival Management and Storage Server) is a file system type which integrates removable media devices (cartridge, tape, etc.), autochanger (robot), and non autochanger configurations. AMASS provides full file, volume, drive and jukebox drive management and the system administration tools needed to control, configure and monitor the jukebox subsystem [AMS94]. Note that to add a new file system at the virtual file system layer does not imply the modification of the host operating system.

In figure 3.1, we can see AMASS file system added to the *virtual file system (vfs)*. Vfs is the file system independent in the UNIX kernel on which other file systems (e.g. UNIX file system (ufs), network file system (nfs), remote file system(rfs), etc.) are attached.

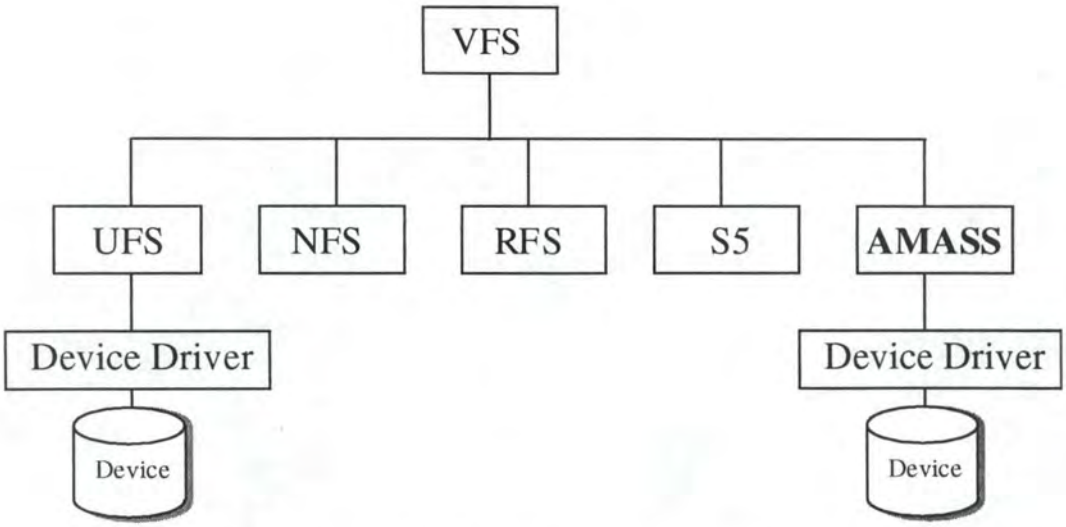


Figure 3.1 AMASS Architecture.

Requirements fulfilled

DataMgr solution satisfies the following requirements :

Automatization : With DataMgr files are automatically migrated from a client site periodically or on demand to maintain available file system space on local disk.

Transparency : DataMgr allows users to be not exposed to the fact of their files are migrated. Users can access to their migrated files as if they were on local disk.

Security : DataMgr preserves security aspects of some files. It creates a Locklist file which is a list of files or pathnames that should be excluded from migration. And also, an other important security aspect provided by DataMgr is the quick restoration at the time of a system crash, in the migration or recall process.

Reliability : DataMgr provides several ways to safeguard migrated files against lost or corruption.

Portability : DataMgr is portable on SUN, IBM RS/6000, HP9000/700 and 800 Series, SVR4, AUSPEX.

Standard : AMASS-DataMgr architecture conforms to the IEEE Mass Storage Reference Model.

HSM level

DatatMgr belongs to level 4 in HSM systems classification.

3.1.1.1. Migration Operations

DataMgr is installed on a file system called *migrating file system.*

Migrating file system is a local file system that has been initialized and enabled for migration and additional functionality supplied by DataMgr.

DataMgr migrates ordinary files and all files within a specified directory, but does not descend into subdirectories. Pipes, special files, symbolic links, sockets are not concerned with migration [DTM94].

Explicit Migration Explicit migration is initiated by the *dmout* command.
Example : *dmout toto.* Where toto is the file to be migrated.

Periodic Migration Periodic Migration is initiated by an administrator job via a cron.

Demand Migration

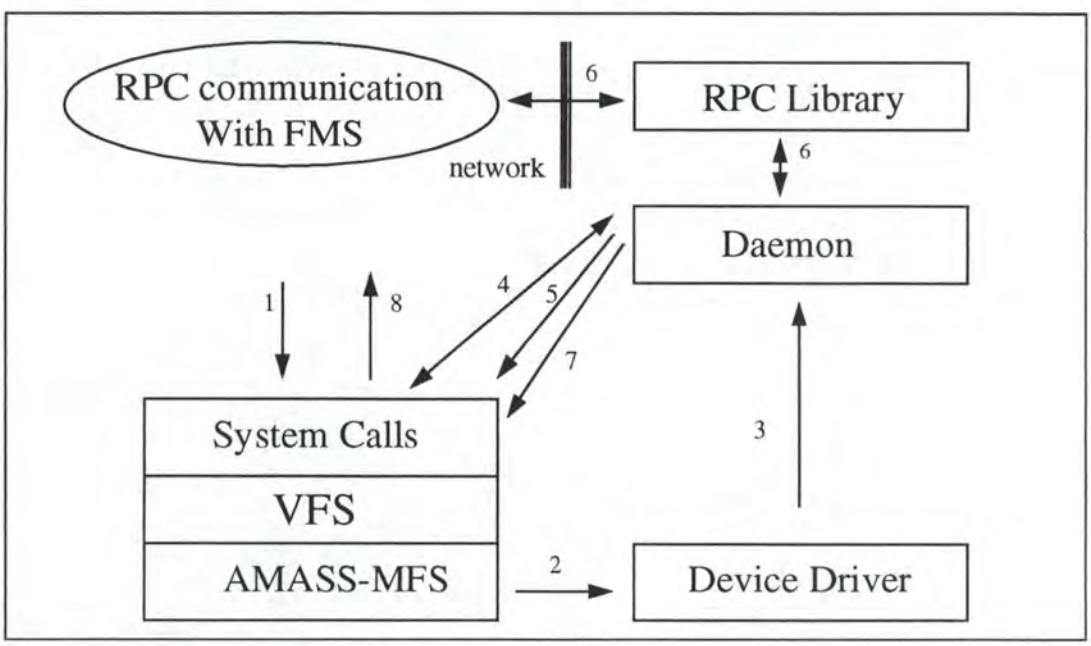


Figure 3.2 Demand Migration Operation.

FMS (File Management Server).
MFS (Migrating File System).

1. An application issues a file system request that requires additional space to be allocated.

2. The DataMgr daemon within the MFS-AMASS (Migrating File System-AMASS) initiates the demand migration operation via the device driver.

3. The daemon within the device driver reads the "no space fault" request.

4. The daemon within the device driver generates a list of all the candidate files that are eligible for migration on the MFS along with their migration attributes (minimum age and size, migration path, etc.).

5. From this candidate files list, the daemon selects a sufficient number of files to be migrated in order to bring the file system utilization percentage down to the selected watermark.

6. For each selected file from the candidates list, the contents are copied to the storage server (File Management Server (FMS)).

7. Once the storage server copy is achieved, the original file on the migrating file system (MFS) is replaced by a stub file (stub file : see below). Files that were migrated are removed from the candidates list.

8. When enough space is available, the original operation proceeds.

A ***stub*** file is what remains of a file on a migrating file system after it has been migrated. A stub file contains the following information :

- The *file leader* is the file containing the first 512 bytes of the migrated file.
- The *file's bitfile id* which is a unique identifier of the migrated file on the storage server. This is a parameter used for the recall operation.
- The *file's logical size* which the size of the file before migrated.

3.1.1.2. Recall Operations

Explicit recall Explicit recall is initiated by the command *dmin.*
Example : *dmin toto,* Where toto is the file to be recalled.

Implicit recall Implicit recall is triggered by a daemon if any of the following cases occurs :

a. Data beyond the file leader (the first 512 bytes) is read.
b. Data is written anywhere within the file.
c. The size of the file is changed (except to truncate to zero length).

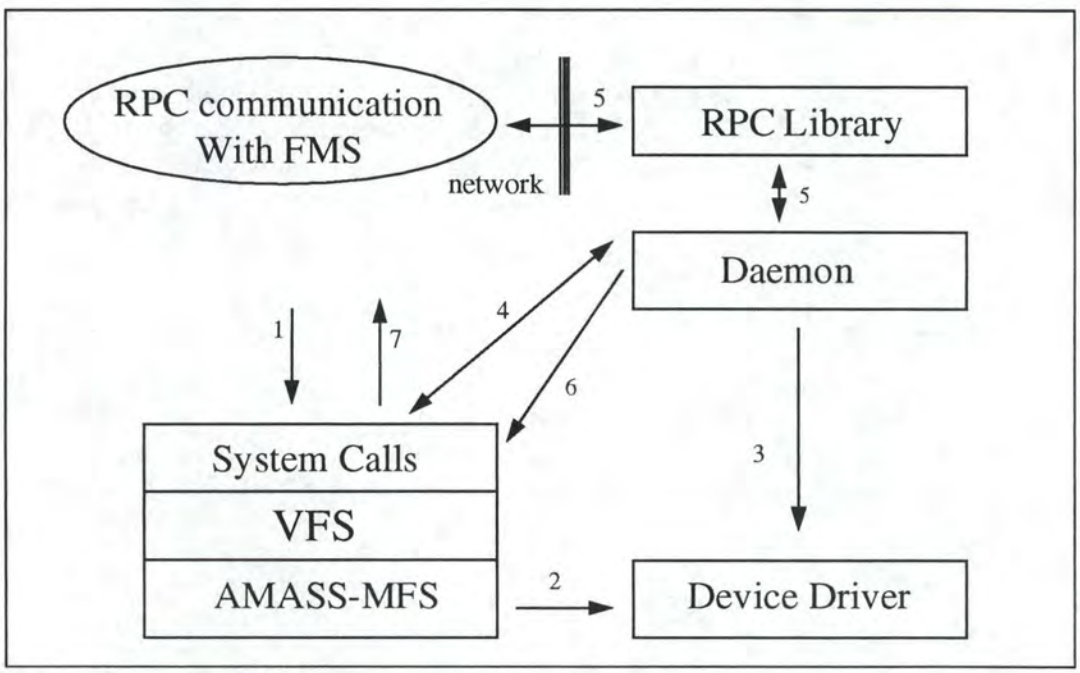


Figure 3.3 Implicit Recall Operation.

1. An application issues an access request to a non-resident file on a migrating file system.

2. The recall daemon is initiated via the device driver.

3. The device driver posts a recall request which is read by the recall daemon.

4. The recall daemon reads the recall request and gets all necessary information (one of them for example is the bitfile ID) from the stub file to retrieve the non-resident file.

5. From the bitfile ID, the recall daemon makes one or more RPC requests to read the associated bitfile and responds within the data.
In any cases (a, b, c, see privious page), the previously non-resident file becomes a shadowed file. And particularly, in case (b), after the bitfile data are reloaded to the local disk, the connection to the old bitfile is broken. A new bitfile ID and a new bitfile will be generated when the modified file is next prestaged or migrated.

6. The recall daemon replaces the non-resident file by a file with the same name containing the recalled data and then deletes the stub file.

7. The original operation can now proceed.

3.1.1.3. Migration Policies

In DataMgr, there are three types of migration : Explicit, Periodic and Demand.
DataMgr performs a migration based on a flexible management policy by which it selects files to migrate and decides when and where to migrate them to. Management Policy includes several factors which are specified by the system administrators. These factors are outlined below.

- Watermarks (high, low).
- File names to be migrated.
- Minimum age.
- Minimum size.
- Factor to weight the age of file to be migrated.
- Factor to weight the size of file to be migrated.
- File management server.
- Migration path.
- Retention time.
- File to be excluded from migration.

Retention time is the length of time a bitfile should be retained on a storage server after its stub file or shadowed file on the storage server has been removed from the client migrating file system.

3.1.2. EpochMigration

EpochMigration is one of the components of Epoch's Data Management solution. In conjunction with Epoch Data Server, EpochMigration provides a file migration solution in distributed environments [EPM92].

Requirements fulfilled

EpochMigration satisfies the following requirements :

Automatization : EpochMigration moves automatically its files (inactive files) to Epoch Data Server. So, time consuming and errors by manually intervention (e.g. identifying files to move) are minimized.

Transparency : EpochMigration allows users to access their "migrated" files as if they were still on local disk.

Portability : EpochMigration is portable on SunOS.

Reliability : With EpochMigration, data on Epoch Data Server are protected.

Standard : EpochMigration solution conforms to the IEEE Mass Storage Reference Model.

HSM level

EpochMigration belongs to level 4 in HSM systems classification.

3.1.2.1. Migration Operations

EpochMigration uses a protocol to perform migration and recall operations. This protocol designed to provide a bitfile service, is a remote procedure.

To manage its data and to avoid to modify the kernel, EpochMigration adds a thin layer between the Virtual File System (vfs) and the UNIX File System (ufs). This layer which consists in a set of modules called *wrappers*, does not replace the existing modules in the operating system. Wrappers intercept file access request and perform two main functions :

1. They determine if the current file accessed is a migrated file, if so manage the recall operation.

2. They allow EpochMigration to intercept the application requiring disk space and, to make more disk space, manage the migration operation.

In addition to these wrappers, EpochMigration uses a pseudo device driver to communicate with migration tasks (workers) , the recall daemon and to reallocate space.

Explicit Migration Explicit migration is initiated by a dedicated command command.

Periodic Migration Periodic migration is initiated by an administrator job, via a cron.

Demand Migration

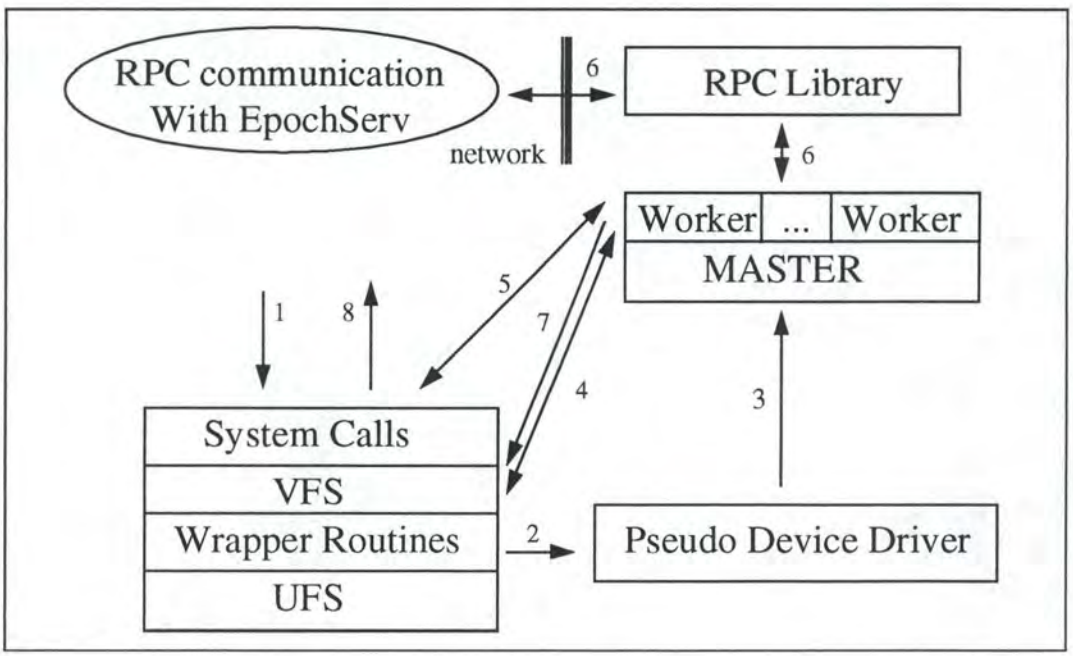


Figure 3.4 Demand Migration Operation.

1. An application issues a file system request that requires additional space to be allocated.

2. The the wrapper routine gets the lower space condition and calls a routine in the pseudo device driver to report it. So, if there is no space available in the file system, the wrapper blocks the operation until space is available; if there is some space available, the operation proceeds immediately, in parallel with the following steps.

3. In either event, the pseudo device driver sends a "no space" fault to the Master. The Master calls the Worker related to that file system.

4. This worker reads inodes to select the best candidates from a policy for migration.

5. The worker reads each file to be migrated.

6. The worker issues one "create bitfile" and one or more "write bitfile" protocol requests for each migrated file.

7. The worker records that this file is migrated to this *bitfile id.*

8. When enough space available, the original operation proceeds.

3.1.2.2. Recall Operations

Explicit recall Explicit recall is initiated a dedicated command command.

Implicit recall

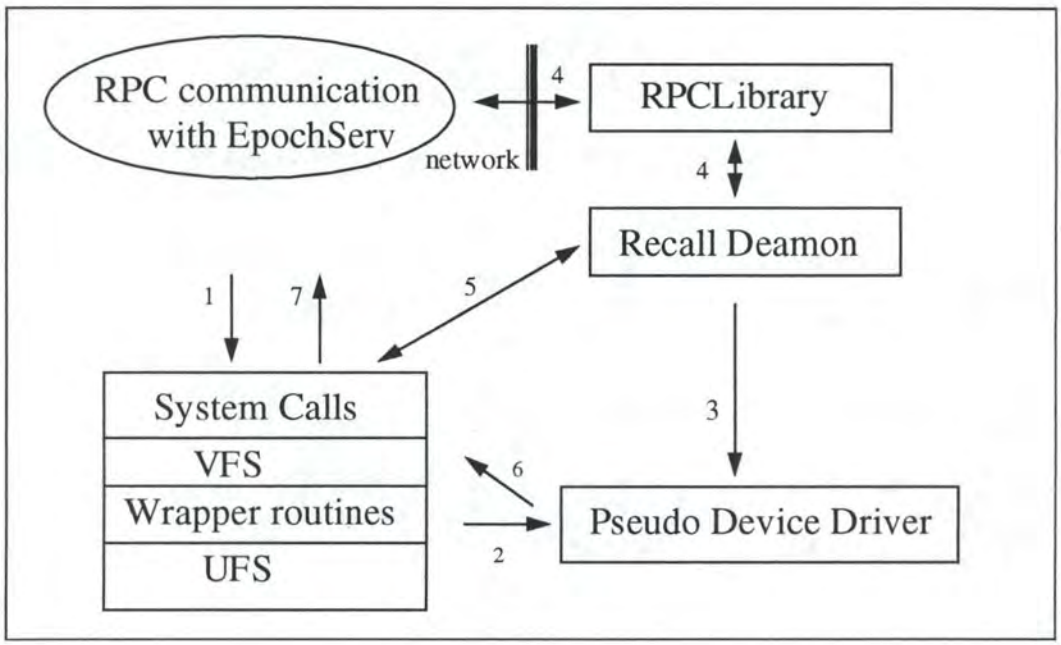


Figure 3.5 Implicit Recall Operation.

1. An application issues a read request to a file.

2. The wrapper intercepts the request, detects if the current file is a migrated file or not. If so, calls a routine in the pseudo device driver and blocks the operation until the entire file is available.

3. The pseudo device driver posts a recall request which is read by the Recall Daemon.

4. The Recall Daemon gets the associated *bitfile id* in the directory containing the migrated file information. It makes one or more RPC requests to the server to read the associated *bitfile* from the *bitfile id* and respond with the bitfile data.

5. The Recall Daemon creates a temporary file containing the recall data and notifies the pseudo device driver the completion of the operation.

6. The pseudo device driver switches the block maps, so the temporary file is now the real file's target.

7. The original operation can now proceed.

3.2. Comparison of the two Products

Basically, there are three differences between the two products.

The first one is related to the architecture. In EpochMigration solution, we have a pseudo device driver which plays a role of interface between workers, recall daemon and wrapper routines. Whereas DataMgr is based on a new file system type added below the vfs layer.

The second one is related to when the recall operation is triggered. In EpochMigration solution when a migrated file is accessed (e.g. read), the entire file is reloaded.
In DataMgr, on the contrary, the recall of the entire file is triggered when only in any of the three following cases occurs :

1. Data beyond the file leader (the first 512 bytes) is read.
2. Data is written anywhere within the file.
3. The size of the file is changed (except to truncate to zero length).

The third difference is related to the requirements fulfilled by each product. A comparative table of requirements fulfilled by each product is shown in figure 3.7.

Requirements		**DataMgr**	**EpochMigration**
Automatization		+ +	+ +
Transparency		-	+
Reliability		?	?
Portability	- Unix	+ +	-
	- Win NT	- -	+
Security	- Secure Transfert	?	?
	- File to be excluded from mig.	+ +	?
Efficiency		?	?
Flexibility	- Migration Policy	+ +	+ +
	- Volume Management	+ +	- -
Standard	IEEE Mass Storage Ref. Model	+ +	+ +

Costs	- Administration	Low	High
	- Realization	High	Low

Figure 3.6 Comparative table from requirements and costs of the two products.

The realization costs of DataMgr solution are higher than EpochMigration's one, developing a new filesystem type requires, for instance, much more person-year than developing a wrapper layer between vfs and ufs.

Chapter 4 Feasibility Analysis

Introduction

This chapter is divided in two parts. In the first one, we will explain some basic aspects of ufs which will allow us to understand the second part of this chapter. In the second part, we will see different solutions to realize implicit recall operation.

4.1. General Aspects of UFS

In this section we describe the basic objects used in this chapter and the followings.

4.1.1. The Virtual File System (VFS)

The kernel's generic abstraction of file system is the file system type independent vfs structure. The vfs structure contains generic information which the kernel uses to manipulate a file system in a file system type independent way. For each active (mounted) file system maintained by the system, an associated vfs structure is allocated and held on a link list. This list is called the *vfs-mount-list* (previously known as the mount-table). The first file system on the *vfs-mount-list* is always a vfs structure named root which, in turn, is referenced by a pointer called *rootvfs*. The *vfs-mount-list* is used by the kernel to administer any file system that are mounted. A file system that is not mounted is not known by the operating system, even if that file system coexists on the same physical device as those file systems that are mounted.

The *rootvfs* contains the information about the root file system that is mounted automatically by the kernel at boot time. Once mounted, the root file system cannot be unmounted unless the system is being shut down. Some fields of the vfs structure are described in figure 4.1.

4.1.2. The Virtual File System Switch Table (vfssw)

To configure a file system type into the operating system, it must have an entry into the Virtual File System Switch table (vfssw).

The vfssw [] table is an array of vfssw structures (see below) each representing a particular file system types. The fields of vfssw are shown at figure 4.1 and an example of a vfssw table is shown at figure 4.2

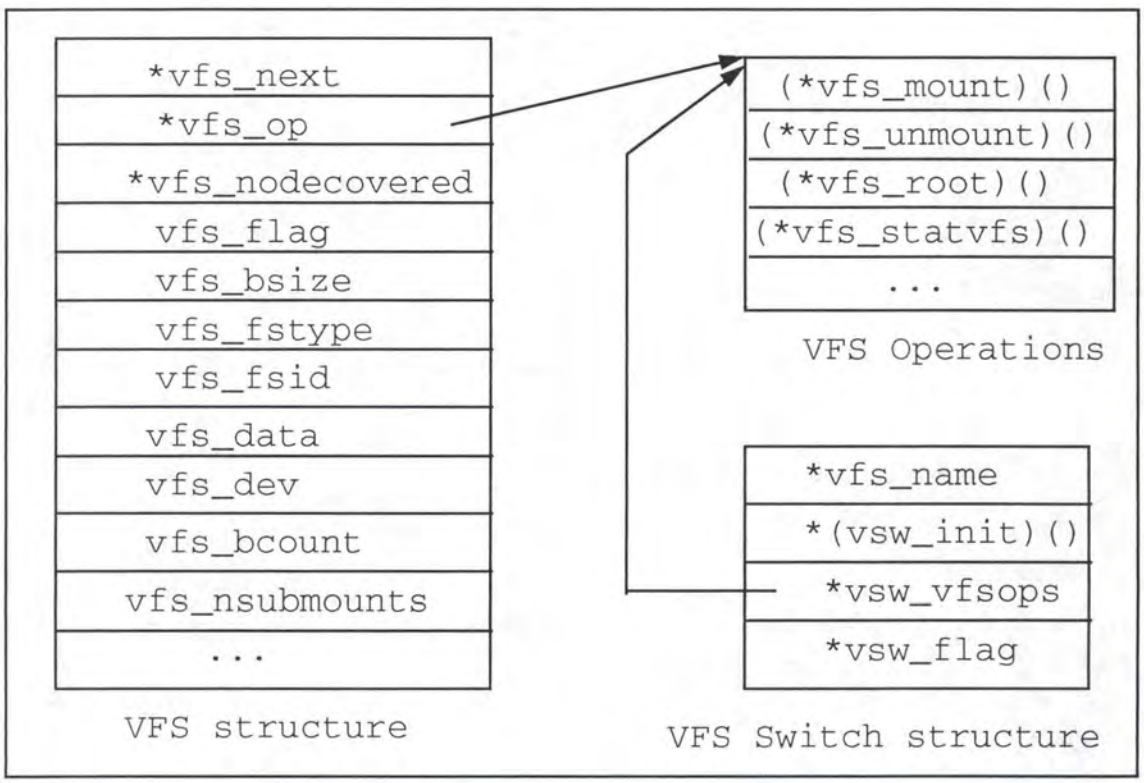


Figure 4.1 vfs and vfsw structures.

vfs_name	vsw_init()	vsw_vfsops	vsw_flag	
0	0	0	0	default values
"spec"	specinit	&spec_vfsops	0	SPEC
"vxfs"	vx_init	&vx_vfsops	0	Veritas
"ufs"	ufsinit	&ufs_vfsops	0	UFS
"nfs"	nfsinit	&nfs_vfsops	0	NFS
"fd"	fdinit	&fdfvfsops	0	FD
"fifo"	fifoinit	&fifovfsops	0	FIFO
"namefs"	nameinit	&nmvfsops	0	NAMEFS
"proc"	prinit	&prvfsops	0	PROC
"s5"	s5ini	&s5_vfsops	0	S5
"rfs"	rf_init	&rf_vfsops	0	RFS
"xnam"	xnaminit	&xnam_vfsops	0	Xenix
"dos"	dosinit	&dos_vfsops	0	MS-DOS

Figure 4.2 Example of a virtual file system switch table.

4.1.3. The Virtual Node (vnode)

The file system independent/dependent split was done just above the UNIX kernel inode layer. This was an obvious choice, as inode was the main object for manipulation of files. By this way, a well defined interface between the two parts can be provided. So, the file system independent inode was named vnode (virtual node) [KLEI86]. While an inode is used to map processes to unix files, a vnode can map a process to an object in any file system type [GRA95].

The vnode was developed in 1984 to abstract the entire file operations in order to support multiple file systems implementation [ROS90]. Vnode is, in fact, a pointer to a file system type specific data structure and functions. Vnodes exist only on memory (not on disk). The fields of a vnode structure is shown at figure 4.3.

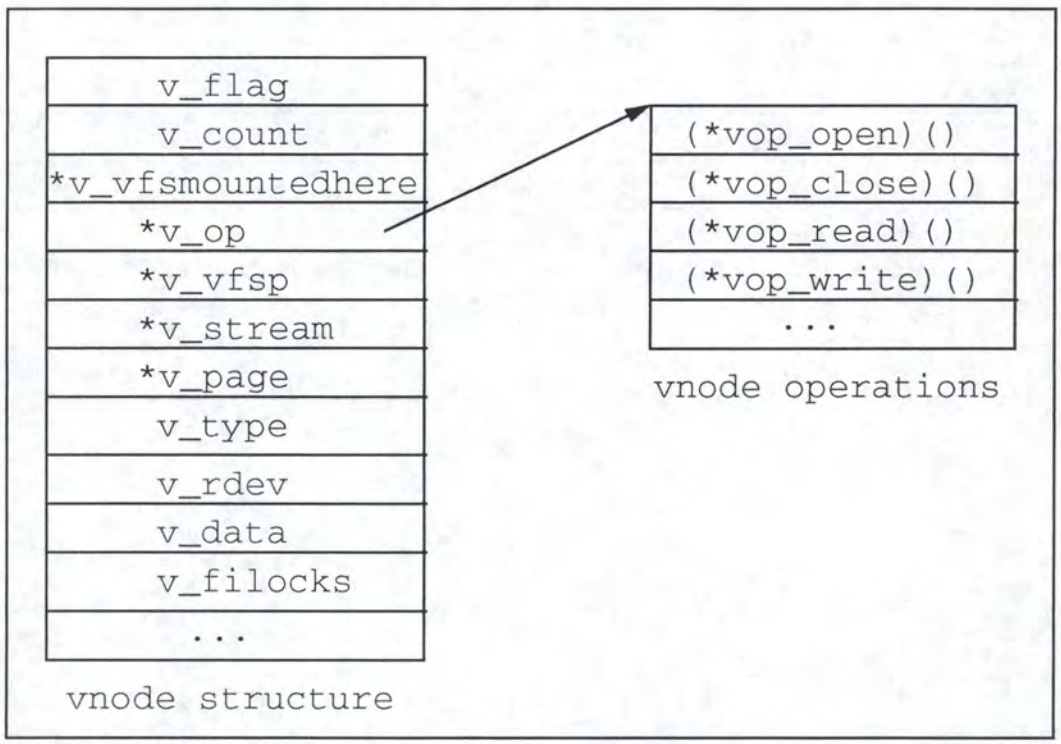


Figure 4.3 Vnode Structure.

4.2. Feasibility Analysis

One of the main problems on which we are faced to build an HSM is related to the implicit recall operation. To realize this type of operation, we need to intercept file access requests. That's why, in this section, we focus to analyse how to intercept file access requests. Since without intercepting file access requests, a read request, for instance, to a migrated file imply an error message because data are no longer on local disk.

There are four ways to intercept file access requests.

The first one is to modify the kernel.

The second one is to replace all ufs routines addresses with our own routine addresses which will allow to intercept file access requests and perform migration and recall functionality.

The third one is to add a thin layer (wrapper layer) between vfs and ufs without modifying the kernel source code.

The fourth one is to add a new file system type into the kernel.

4.2.1. Modify the Kernel Source Code

This is the worst solution for two main reasons. First, it supposes to have the kernel source code and the second one, it is, specially, a non portable solution. This solution is therefore not considered.

4.2.2. Replace all ufs-related Addresses

In this case there are two methods.

1. The first one depends upon the version of UNIX. It consists to look into where the vfssw (Virtual File System SWitch) table is generated and replace the entries there. So, this method is not portable.

2. The second one supposes at first to have the kernel source code. It consists to manipulate the routine addresses in the global data structure *« struct vnodeops ufs_vnodeops »* (see figure 4.4) declared in the *ufs_vnops.c* file by replacing all ufs-related addresses with our own routine addresses.

The global data *« struct vnodeops ufs_vnodeops »* contains all addresses of ufs specific routines used by VOP_* macro declared in vnode.h. And these routines are activated by the vnode layer.

/* ufs_vnodeops.c */

```
struct vnodeops ufs_vnodeops = {
      ufs_open,          /* open */
      ufs_read,          /* read */
      ufs_write,         /* write */
      ufs_close,         /* close */
      ...
};
```

Figure 4.4 Global data structure for ufs in ufs_vnodeops.c.

4.2.3. Add a Wrapper Layer Between vfs and ufs

To add a wrapper layer between vfs and ufs can be achieved by generating a new "pseudo" file system type; this pseudo file system type offers an initialization routine which manipulates the ufs_vnodeops-addresses within the ufs-inodes (these are assigned by ufsinit() during startup). The file system-initialization routines are called in alphabetical order, the wrapper initialization must be activated **after** ufsinit(), so we have to choose a name for the pseudo file system type that guarantees that the appropriate initialization routine is called after ufsinit() (for example "zfs").

After the pseudo file system is configured and linked to the UNIX kernel, the zfs_*()-routines (wrapper routines) are now called by the vfs layer; the wrapper routine have to call the appropriate ufs_*() routines, example (see figure 4.5).

Note that this way does not imply the modification of the kernel source code.

Example

```
zfs_read(vp,uiop,ioflag,cr)
      struct vnode *vp;
      struct uio *uiop;
      int ioflag;
      struct cred *cr;
{  int error;

      ...
      /* our own code */
      /* example */
      printf(" we are in read access!!!\n");
      ...
      error=ufs_read(vp,uiop,ioflag,cr);
      ...
}
```

Figure 4.5 Example of a read wrapper routine.

4.2.4. Add a New File System Type

The third way is to add a new file system type into the kernel. With this way we extend the native operating system. This solution is portable in all Unix platforms.

When the new file system is created, then it can be mounted via the "mount - F <type>" option or by specifying the proper option in the /etc/vfstab file (an example is shown at figure 4.6).

special	fsckdev	mountp	fstype	ckpass	automnt	mntflags
/dev/dsk/c0d0s1	/dev/root	/	ufs	1	yes	-
/dev/dsk/c0dd0s2	/dev/rdsk/c0dd0ss5	/user	s5	-	no	rw
/proc	-	/proc	proc	-	no	-
		...				

Figure 4.6 Example of vfstab.

The two best solutions seem to be the two last ones. The solution which consists to add a wrapper layer between vfs and ufs will be used in the following chapters.

Chapter 5 Integration of Migration/Recall into HSMS/HSMS-CL

Introduction

In the previous chapters, we have seen the general aspects of HSM. We have also analysed different solutions about file migration. In this one, we will see how to integrate these concepts, particularly the file migration and the recall functionality in HSMS/HSMS-CL product. Since the integration of a migration and recall depends closely on the chosen solution, we have proposed the one which add a thin layer between vfs and ufs. Why this solution ? It is mainly for costs realization reason and for this work, it is the easiest one for the prototype design described in the following chapter.

5.1. Administrator's View

In this section we will see the administrator's view at client side and server side. On client side we will describe only the administration for active clients.

5.1.1. Administration on Client Side

5.1.1.1. Migration Policy File

The administrator of the workstation on which HSMS-CL is installed has to initialize the migration policy file specifying the migration policy attributes. After initialization, the administrator is the only one who can modify the migration policy file (e.g. modification of watermarks).

5.1.1.2. Inhibited Migration File

Inhibited migration file is a file which contains pathnames of files that should be excluded from migration.

Here, we separate the files that should be excluded from migration in two categories.

In the first, we have the files that should be excluded from migration in any circumstances. These files are concerned with the bootstrap file, configuration files, etc.
Pathnames of these files are inserted in what we call the static inhibited migration file.

The second category is concerned with the user's inhibited migration files (i.e. to allow a user the possibility to exclude some of his own files from migration). These files are inserted, by giving the pathnames, in what we call the dynamic inhibited migration file.

So, a particular user can add an entry or delete his entry (i.e. an entry added by him) to/from the dynamic inhibited migration file. The administrator of the workstation has to define for each user a quota in term of number of bytes of files data for the dynamic inhibited migrated file.

Example : Let's assume for a user toto who has a quota of X MB for the dynamic inhibited migration file. The number of bytes of all the toto's files which have an entry in the dynamic inhibited migration file is always less or equal than X MB.

Note : The entries in the dynamic inhibited migration file will not be migrated during periodic or demand migration runs, but can still be migrated explicitly by user request via a dedicated command (see *bsmig*). In this later case, after explicit migration, the related entry is deleted from the dynamic inhibited migration file.

5.1.2. Administration on Server Side

5.1.2.1 Optimizing Media

As successive recalls and deletes occur, progressively more and more of the tape will contain wasted space. So, it is necessary to reorganize media or to optimize it.

Media reorganization is a three step process that reclaim wasted space on the medium to optimize the media space available to user, but also to improve access time for retrieval migrated files. We describe below the following steps to reorganize media.

step 1 Identify only the migrated files from the file system.

step 2 Copy only the migrated files to another medium.

step 3 When the copy is stable, the original medium, if empty, is labelled blank for a next usage.

Media reorganization can be made periodically via a cron or by the administrator command COPY-NODE-SAVE-FILE.

5.1.2.2. Migration on Storage Levels

As excepted the first storage level, all others are on the storage server, and to conform to HSM concept, the system administrator must have the possibility to migrate files explicitly and periodically from a storage level i to a storage level j such i<j.

Since we suppose that space is always available on the storage server, explicit and periodic migration are sufficient.

5.1.2.3. Accounting for Workstation Files

Accounting is a processing that generates a bill that contains the quantity of resources and services used by an end user.

Since there are different types of file on the server (migrated files, backed up files, etc.), it is interesting to create an accounting for each type of file. But here, we will focus only to the accounting for migrated files.

In the storage server, there is two ways to process accounting :

1. **Periodic Accounting** : account records are written when requested (by an administrator command or a permanent recurring task) and then evaluated afterwards with user program. This accounting can only concern "static" resources like media space.

2. **Permanent Accounting** : account records are written while a task uses resources. These account records must be evaluated afterwards by a user program. The main disadvantage is that additions must be done on several account records to obtain the final result. That's why in general the customer requests only the periodic accounting.

What kind of resources must be accounted during the server run ?

1. CPU usage : Due to the high usage of the network, the CPU utilization is high during migration/recall runs.

2. TAPE/CARTRIDGE space : This is a main resource used by the server to store migrated/backed up/archived files.

3. DISK space : Due to the great number of files that are processed by the server, the migrated/backed up files and the repository use a huge amount of space in S1 level.

Who must pay for resources consumption ?

Since migration services can be offered in the network to all users who have HSMS-CL, it is normal that the end user pays for this service. Thus, the possibility to send a bill to the end user must be offered to the system administrator.

Periodic Accounting

To perform a periodic accounting on S1 level, we need : Per workstation/user : the total number of migrated files' bytes on the tapes. As this kind of information is recorded into the REPOSITORY files, a scan of all 'REPOSITORY migrated files' is sufficient.

To perform a periodic accounting on DISK space used by migrated files and the REPOSITORY, it is also sufficient to scan all REPOSITORY migrated files of a workstation and to compute the number of bytes used.

This accounting type can be triggered either by an HSMS administrator command or by a permanent recurring task dedicated for this purpose.

Permanent Accounting

Due to the task structure of HSMS/ARCHIVE (HSMS communication task, HSMS server task, ARCHIVE-main-task, ARCHIVE-subtask, HSMS-answer-task), to realize a permanent accounting (of CPU and DISK space resources), it is necessary to implement hooks in the code of HSMS/ARCHIVE.

Per workstation/user these hooks must be able to :

1. Add CPU consumption to a previous value.

2. Add DISK space usage to a previous value.

3. Write accounting records when the processing is finished or when it is no more possible to maintain information into a buffer.

Where to implement these hooks ?

These hooks must be implemented at the following places :

1. In the HSMS communication task to account the CPU time consumption of an incoming request.

2. In the HSMS server task to account DISK space used by the HSMS reporting processing.

3. In the ARCHIVE-main-task to account the CPU time consumption of the « parsing process » phase and the DISK space used by the reporting.

4. In the HSMS-answer-task to account the CPU time and DISK space used by the processing that send report to the workstation.

It is proposed to activate this accounting type only if it is requested by the HSMS administrator. Main reasons are :

1. The periodic accounting type is sufficient if the values of CPU time used for reporting are not requested.

2. The path length of the whole processing is increased.

3. The increase of the accounting records.

One of the big advantages of the accounting is that it allows to the administrator of a workstation to have a way (analysing accounting) to reorganize his migration policy.

5.2. User's View

In this section, we will describe user's view for only active clients (i.e. clients which have the HSMS-CL product).

We have three type of interface :

- Command Line Interface (CLI).
- Graphical User Interface (GUI).
- Application Programmer Interface (API).

5.2.1. Command Line Interface

This part is concerned with the addition of two new commands into the existing HSMS-CL product. The migration command and the recall command.

Migration Command

NAME

bsmig - migrate files-.

SYNOPSIS

bsmig
[-(i | include) <path>...]
[-(x | exclude) <path>...]
[-(nr | noreport)]
[-(r | report)]
[-(help)]

DESCRIPTION

-i, -include

This option allows non-privileged users and privileged users (super user) to migrate files.

You must be a super user or the file owner. You can only migrate regular files (directories, FIFOs, special files, symbolic links are not concerned with migration).

You cannot migrate files with an *atime* less than the minimum age defined in the migration policy file. You cannot migrate file smaller than the minimum size defined in the migration policy file.

-x, -exclude
This option allows specification of a path or a set of paths for the workstation's files **not** to be migrated.

-r, -report
With this option, a full report of what has been actually migrated is generated.

-nr, -noreport
no report is generated.

-help
This option forces only the help page to be displayed.

Example bsmig -i /home34/toto/* -x test.c -r
migrate all files within the /home34/toto excepted the file test.c and return a full report.

Recall Command

NAME

bsrec - recall files-.

SYNOPSIS

bsrec
[-(i | include) <path>...]
[-(x | exclude) <path>...]
[-(nr | noreport)]
[-(r | report)]
[-(help)]

DESCRIPTION

-i, -include

This option allows non-privileged users and privileged users (super user) to recall files.

You must be a super user or the file owner to make a recall.

-x, -exclude
This option allows specification of a path or a set of paths for the workstation's files **not** to be recalled.

-r, -report
With this option, a full report of what has been actually recalled is generated.

-nr, -noreport
no report is generated.

-help
This option forces only the help page to be displayed.

Example bsrec -i /home34/toto/example -nr
recall the file /home34/example without generating a report.

5.2.2. Graphical User Interface

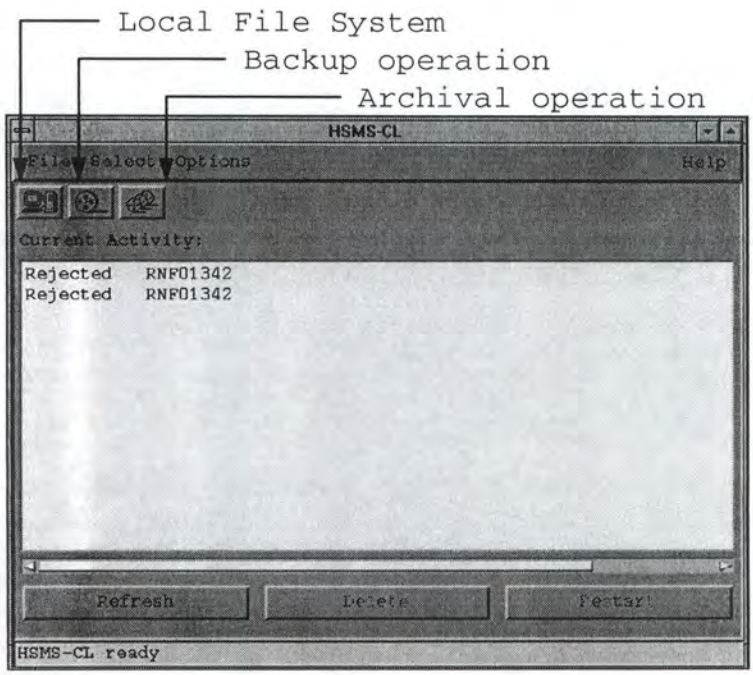


Figure 5.1 HSMS-CL main screen.

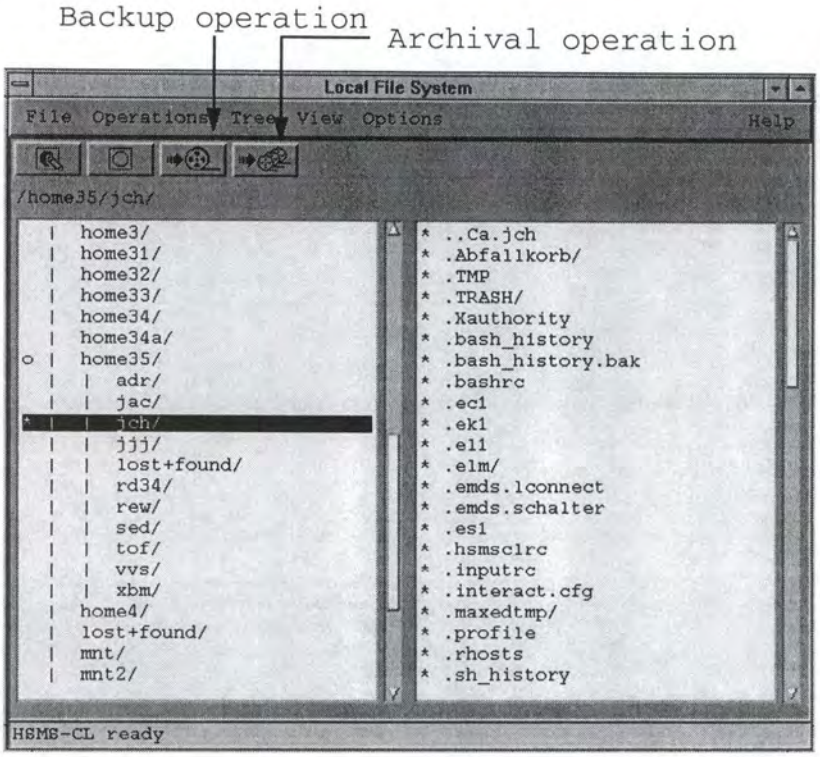


Figure 5.2 Existing backup and archival interface.

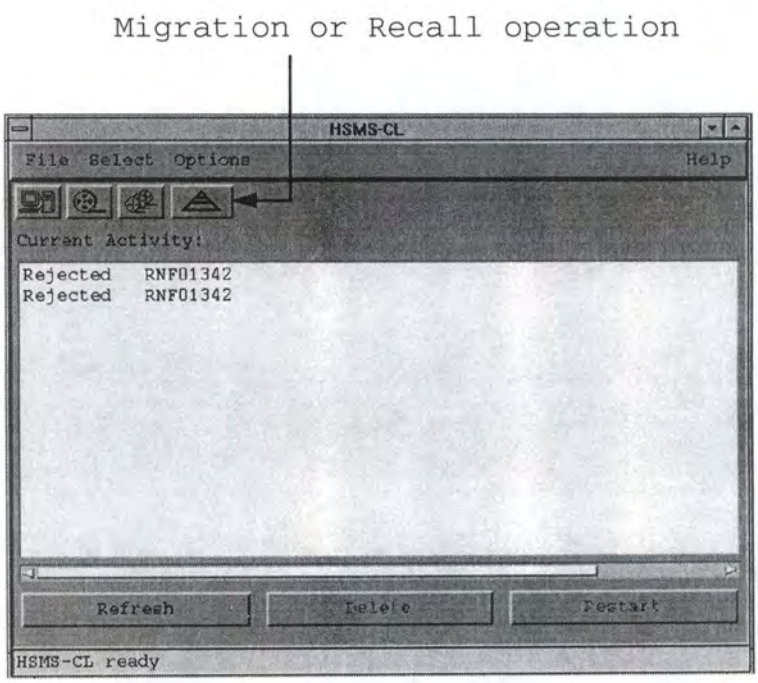


Figure 5.3 HSMS-CL main screen interface proposal.

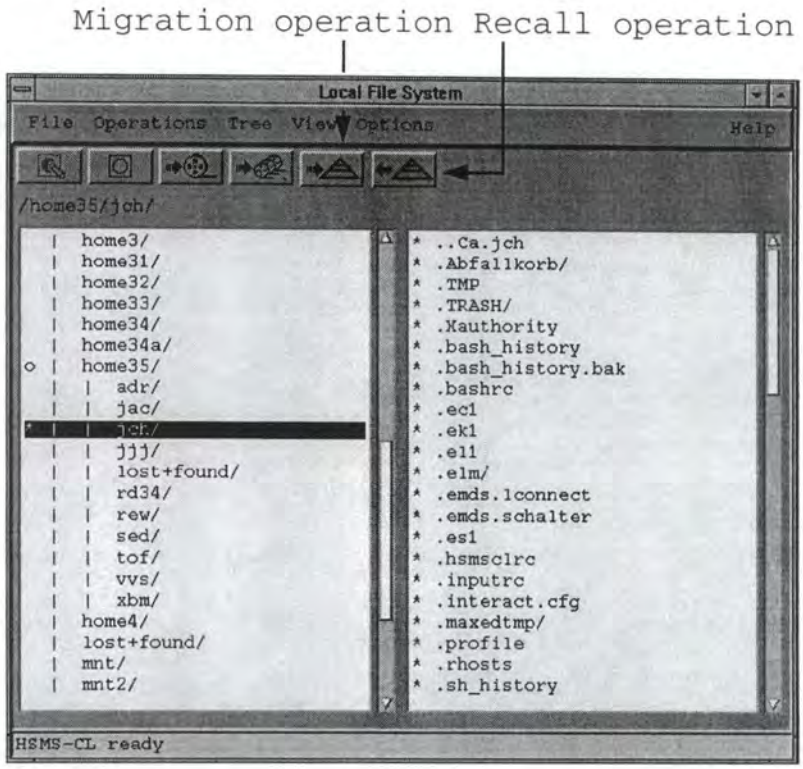


Figure 5.4 The two new functionality interface proposal.

5.2.3. Application Programmer Interface

The HSMS-CL Application Programmer Interface (API) allows programmers to access the features of HSMS-CL with a C programming language.

Working with HSMS-CL API requires knowledge of the HSMS-CL functionality and the C programming language of the UNIX operating system.

In the existing HSMS-CL API, we propose to add two following functions :

HSMSCL_mig (for the migration) and HSMSCL_rec (for the recall).

5.3. Architecture supporting Migration Functionality

In this section, we describe the three migration taskings that we propose in this work which are :

- Explicit migration tasking,
- Demand migration tasking,
- Periodic migration tasking.

5.3.1. Explicit Migration Tasking

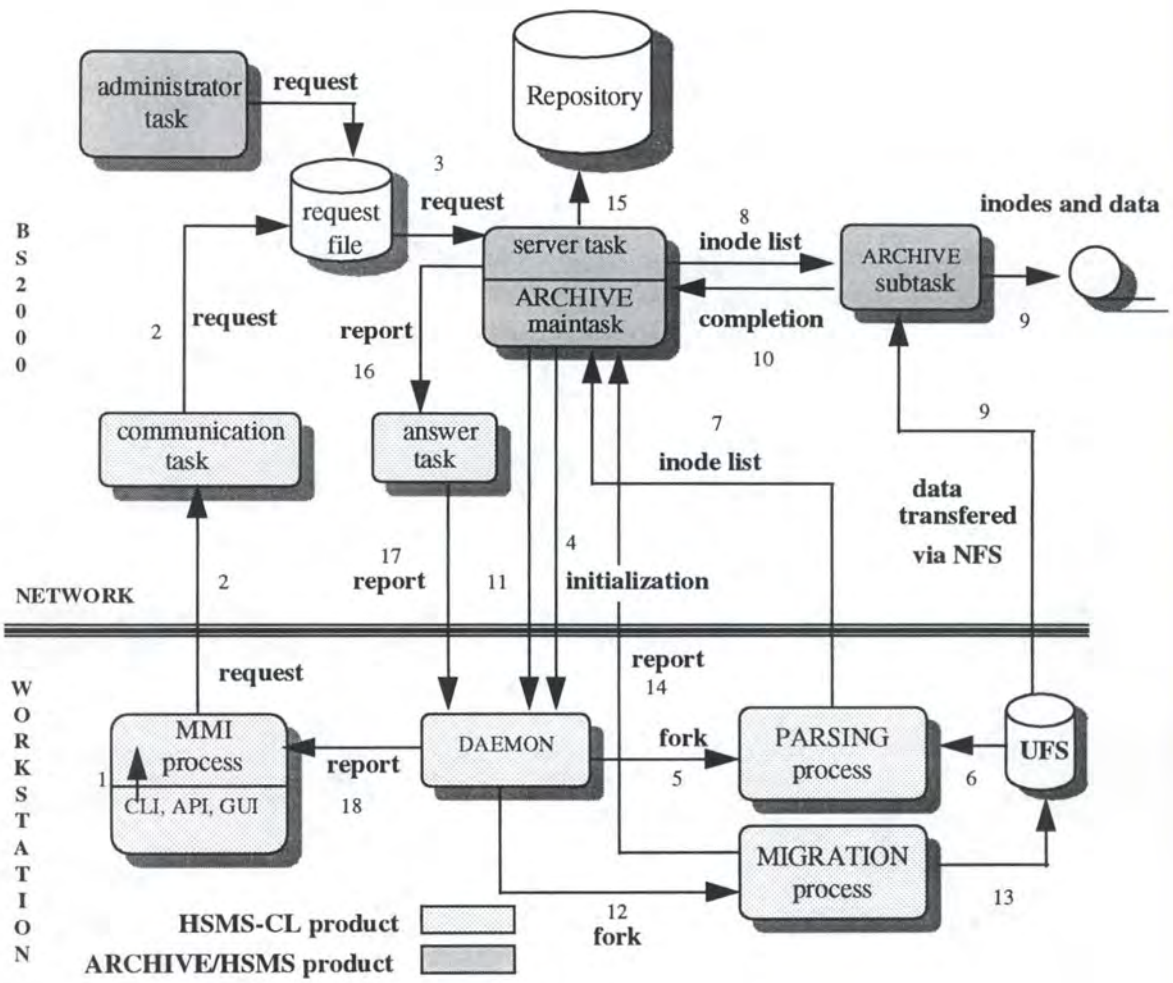


Figure 5.5 Explicit Migration Tasking.

1. A user or a system administrator issues a request to migrate one or many files via a dedicated command.

2. The migration request passes through the COMMUNICATION TASK which sends it to the REQUEST FILE.

3. A request is sent to a HSMS-Server-Task. The first free Server Task gets the request.

4. The ARCHIVE MAIN TASK issues an initialization request to the DAEMON.

5. The DAEMON creates a PARSING process.

6. The PARSING process gets the related inodes from the UFS.

7. The PARSING process sends the inode list to ARCHIVE MAIN TASK.

8. ARCHIVE MAIN TASK creates one or many subtasks.

9. ARCHIVE SUBTASK processes the transfer of the data on a save file (e.g. on tapes).

10. ARCHIVE SUBTASK advises the ARCHIVE MAIN TASK (that is the SERVER TASK) when the process is completed and returns a bitfile ID for each migrated file.

11. ARCHIVE MAIN TASK sends an initialization request to, the DAEMON giving the inode list.

12. The DAEMON creates a MIGRATION process.

13. The MIGRATION process deletes data related to inodes from the inode list.

14. The Migration process sends a report to the ARCHIVE MAIN TASK.

15. The SERVER TASK updates the REPOSITORY recording all necessary information (Volume Serial number, Block Number, Sequence/Block Number, Block ID) to retrieve migrated files.

16. The report is transmitted to the ANSWER TASK by the SERVER TASK.

17. The report is transmitted by the ANSWER TASK to the DAEMON.

18. The DAEMON sends the report to the MMI (Man Machine Interface) process and signals that the migration has succeeded or not.

5.3.2. Demand Migration Tasking

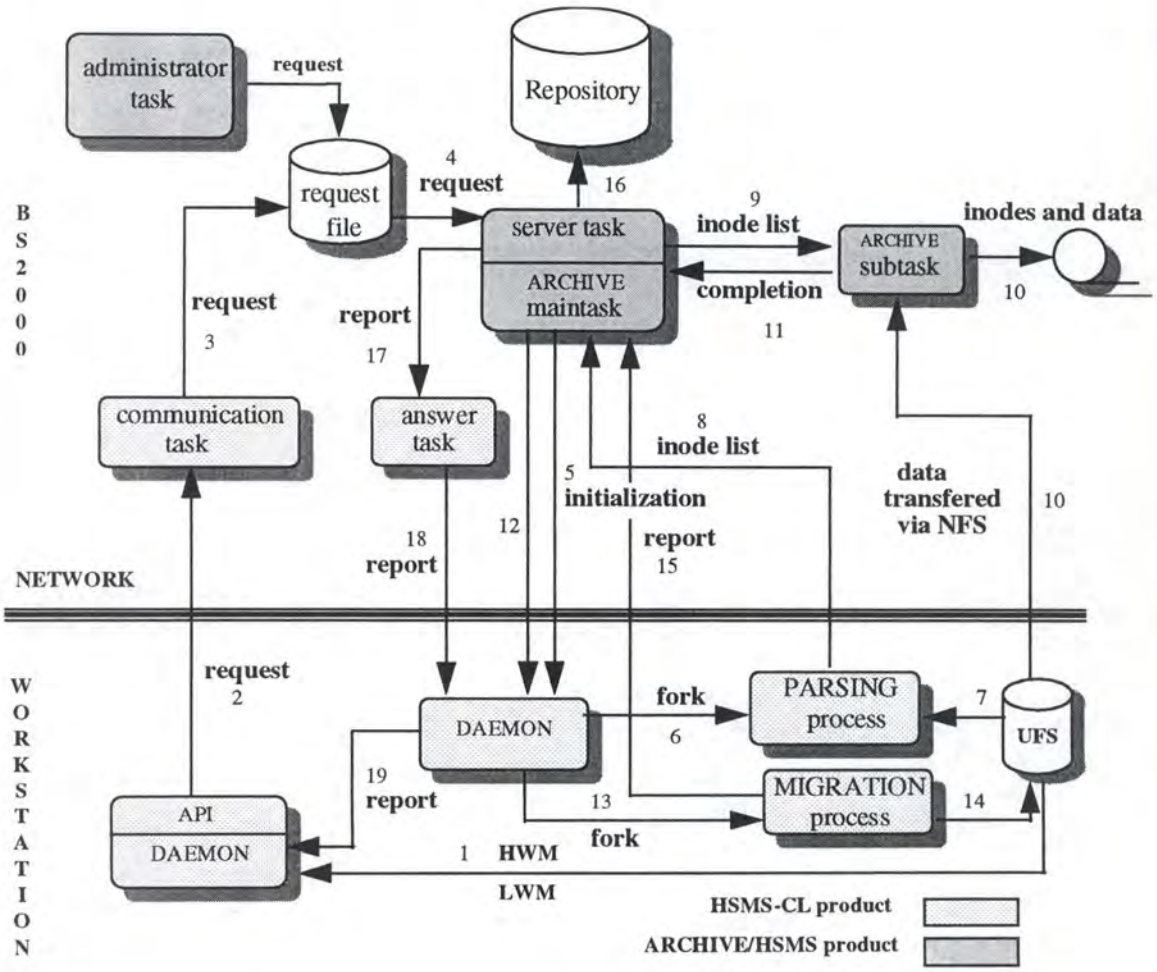


Figure 5.6 Demand Migration.

1. When the HWM (High Watermark) is reached, the DAEMON gets the low space condition (LWM).

2. The DAEMON calls a routine in the Aplication Programmer Interface (API) that issues a request to the COMMUNICATION TASK.

3. The migration request passes through the COMMUNICATION TASK which sends it to the REQUEST FILE.

4. If a server task for a migration archive is available, the request is sent. If not this request waits and the others in the request file are handled.

5. The ARCHIVE MAIN TASK issues an initialization request to the DAEMON.

6. The DAEMON creates a PARSING process.

7. The PARSING process gets the related inodes of canditate files for migration from the UFS.

8. The PARSING process sends the inode list to ARCHIVE MAIN TASK.

9. ARCHIVE MAIN TASK creates one or many subtasks.

10. ARCHIVE SUBTASK processes the transfer of the data on a save file (e.g. on tapes).

11. ARCHIVE SUBTASK advises the ARCHIVE MAIN TASK (that is the SERVER TASK) when the process is completed and returns a bitfile ID for each migrated file.

12. ARCHIVE MAIN TASK sends an initialization request to, the DAEMON giving the inode list.

13. The DAEMON creates a MIGRATION process.

14. The MIGRATION process deletes data related to inodes from the inode list.

15. The Migration process sends a report to the ARCHIVE MAIN TASK.

16. The SERVER TASK updates the REPOSITORY recording all necessary information (Volume Serial number, Block Number, Sequence/Block Number, Block ID) to retrieve migrated files.

17. The report is transmitted to the ANSWER TASK by the SERVER TASK.

18. The report is transmitted by the ANSWER TASK to the DAEMON.

19. The DAEMON sends the report to demand migration DAEMON and signals that the demand migration has succeeded or not.

5.3.3. Periodic Migration Tasking

Periodic migration tasking is the same as the demand migration tasking excepted that is initiated by an administrator job (e.g. via a cron).

5.4. Architecture Supporting Recall Functionality

In this section, we describe the two recall taskings that we propose in this work which are :

- Implicit recall tasking,
- Explicit recall tasking.

5.4.1. Implicit Recall Tasking

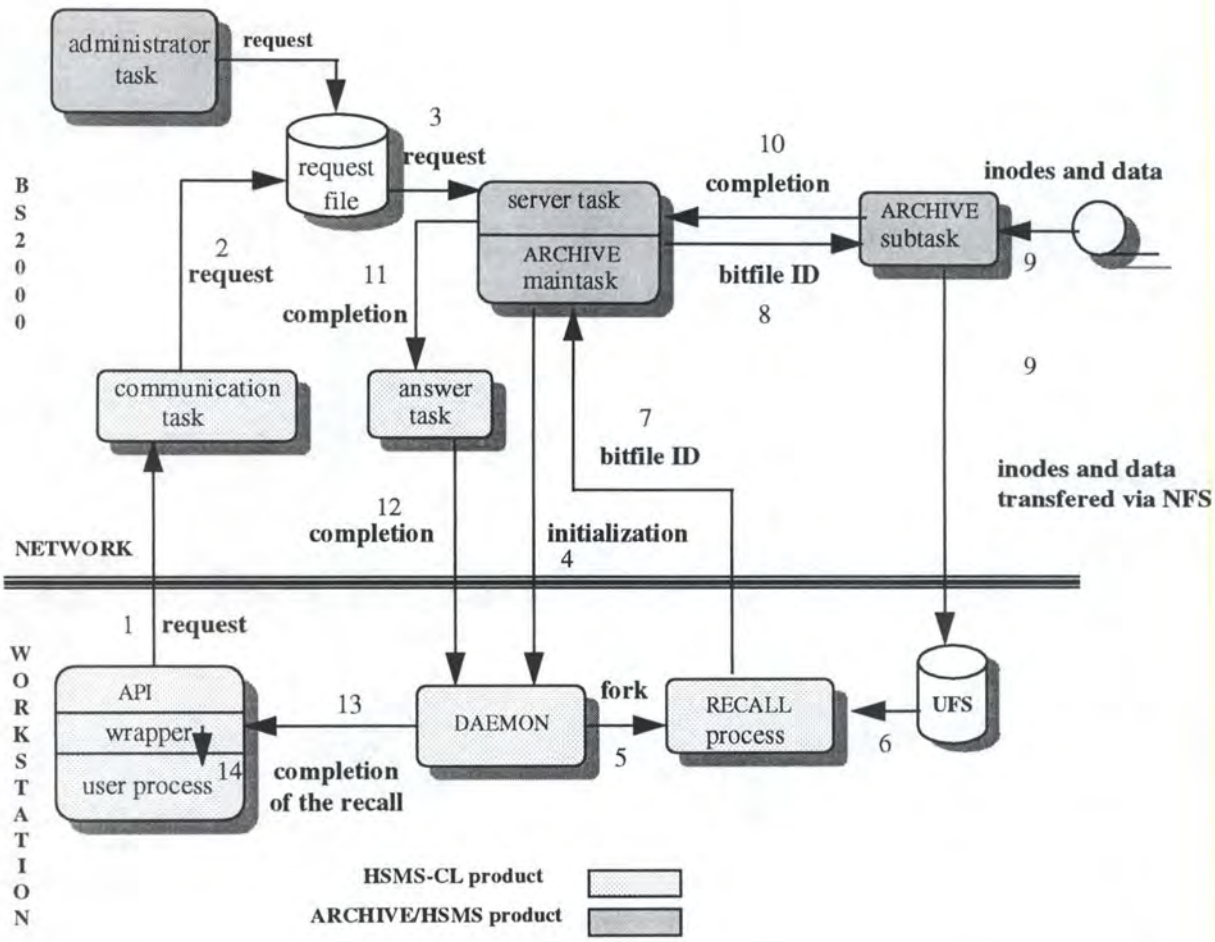


Figure 5.7 Implicit Recall.

1. When a user or an administrator of a workstation issues a request to access to a migrated file, the appropriated wrapper routine intercepts the access and calls HSMS_CL API to send the request to the COMMUNICATION TASK.

2. By the COMMUNICATION TASK, the request is sent to the REQUEST FILE

3. If a SERVER TASK is available, the request is sent to it.

4. The SERVER TASK, which is now the ARCHIVE MAIN TASK, issues an initialization request the DAEMON.

5. The DAEMON creates a RECALL process.

6. The RECALL process gets the associated bitfile ID.

7. The RECALL process sends the bitfile ID to the ARCHIVE MAIN TASK.

8. ARCHIVE MAIN TASK creates one or many subtasks and sends the bitfile ID.

9. The ARCHIVE SUBTASK finds the bitfile, from the bitfile ID and other information, and responds with the bitfile data via NFS.

10. When the transfer is completed, ARCHIVE SUBTASK terminates the SERVER TASK.

11. The SERVER TASK notifies the ANSWER TASK the completion of the recall.

12. The ANSWER TASK notifies the DAEMON the completion of the recall.

13. The DAEMON advices the wrapper routine the completion of the recall to signal if the recall has succeeded or not.

14. The wrapper calls UFS.

5.4.2. Explicit Recall Tasking

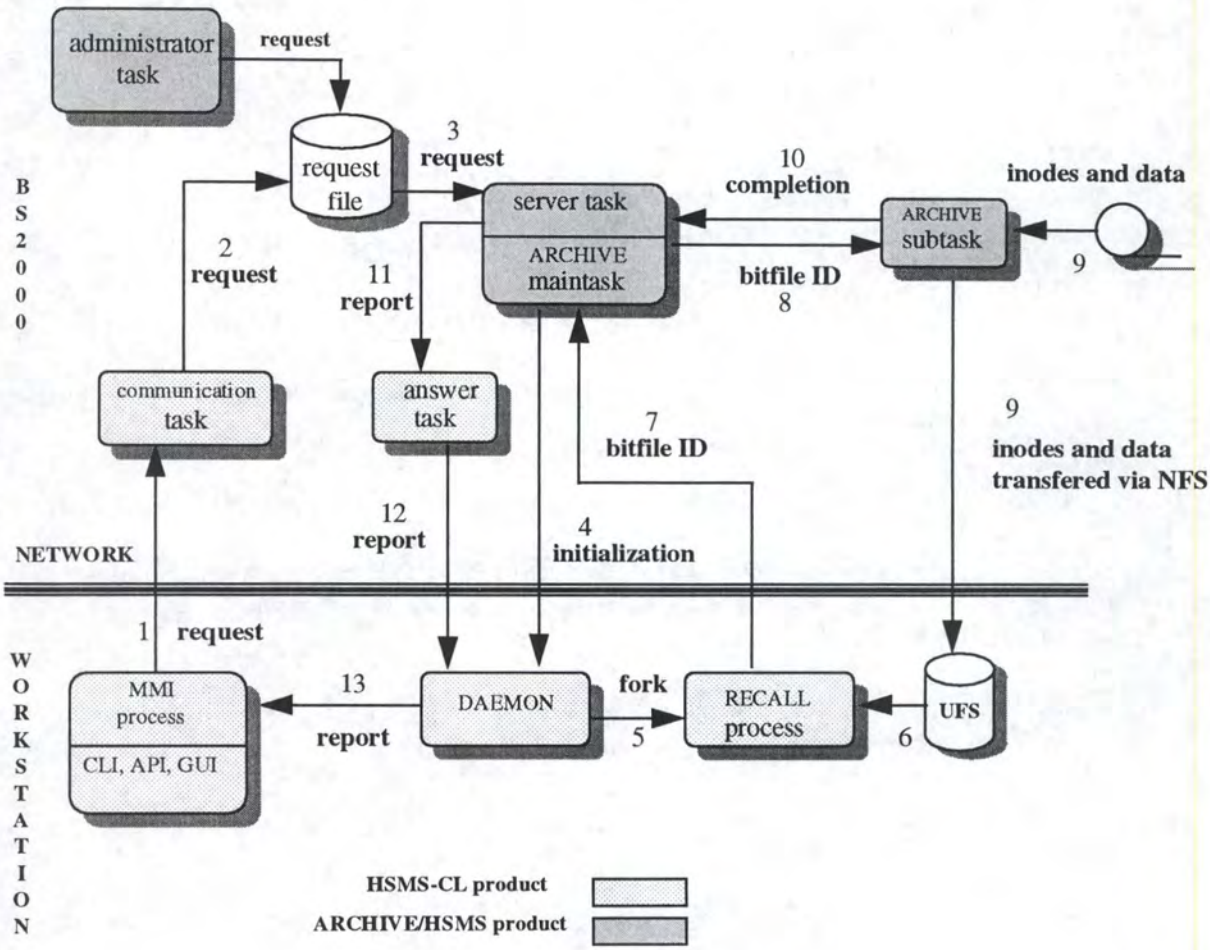


Figure 5.8 Explicit Recall Tasking.

The explicit recall steps are the same as the implicit recall steps excepted the step 1 and the step 13. At the step 1 the explicit recall is initiated by the bsrec command. At the step 13 the DAEMON sends in addition a report to the MMI process.

Chapter 6 Prototyping

Introduction

In this chapter, we present a prototype based on the study done at the previous chapters. This prototype is divided in four parts.

1. bls command,
2. Migration functionality,
3. Recall functionality,
4. Intercepting file access request.

6.1. bls Command

This command allows to an user to distinguish the resident files and the non-resident files. As here, a migrated file is replaced by an empty file but with a bittfileID in the */tmp/mif/fileid* directory, it is necessary to do the difference between a migrated file and an empty file.

By *bls* command, we avoid to modify the *ls* command. With *bls* command, we can see, for a migrated file, a m bit and the length of the file before migrated in order to do the difference between resident and non-resident files (see figure 6.2). If there is no migrated file, *bls* command has the same effect as the *ls* command with the *-l* option(see figure 6.1).

```
sba~>ls -l
total 576
-rwx------   1 sba      RD34         4925 Jan  3 10:07 -g
drwx------   3 sba      RD34          512 Sep 11 16:38 ICONS
drwx------   2 sba      RD34          512 Jan 22 13:52 Mail
-rwx------   1 sba      RD34         4085 Jan  4 11:20 a.out
-rwx------   1 sba      RD34         4585 Jan 24 14:50 acc
sba~>bls
total 576
-rwx------   1 sba      RD34         4925 Jan  3 10:07 -g
drwx------   3 sba      RD34          512 Sep 11 16:38 ICONS
drwx------   2 sba      RD34          512 Jan 22 13:52 Mail
-rwx------   1 sba      RD34         4085 Jan  4 11:20 a.out
-rwx------   1 sba      RD34         4585 Jan 24 14:50 acc
```

Figure 6.1 ls and bsl commands.

6.2. Migration Functionality

In this part, we show a simple way to realize a migration functionality and particularly some important aspects to take into account.

These aspects are related, for instance, to the files to be excluded from migration, security (e.g. a user can migrate only his own file), etc.

Only explicit migration operation is implemented here (demand and periodic migration are not realized). The explicit migration operation is initiated via a command that we have called here *bsmig*.

Migration Policy

The Which

- A user can migrate only his own files.
- An empty file is not migrated.
- Files to be excluded from migration have their pathname inserted in the directory */home34/sba/excluded.*

The When

Since demand and/or periodic migration operations are not realized here, this policy is not taken into account.

The Where

We have chosen the directory */tmp/mig* where migrated files reside.

bsmig command

bsmig filename.

Example :

bsmig /home34/sba/toto, migrates the file */home34/sba/toto* to */tmp/mig*.

*bsmig se**, migrates all files from the working directory whose file names start by *se*.

If the migration of a specified file has succeeded, the migration operation returns a *bitfileID* in the */tmp/mig/fileid* directory. This bitfileID contains the real pathname of the migrated file (see figure 6.2). In UNIX, as the inode number and the device number of a file identify it uniquely, we have chosen the *bitfileID* as the concatenation of the inode number and the device number. After migration, the original file is truncate to zero length i.e. replaced by an empty file with the same name, but with a bitfile ID in /tmp/mig/fileid directory.

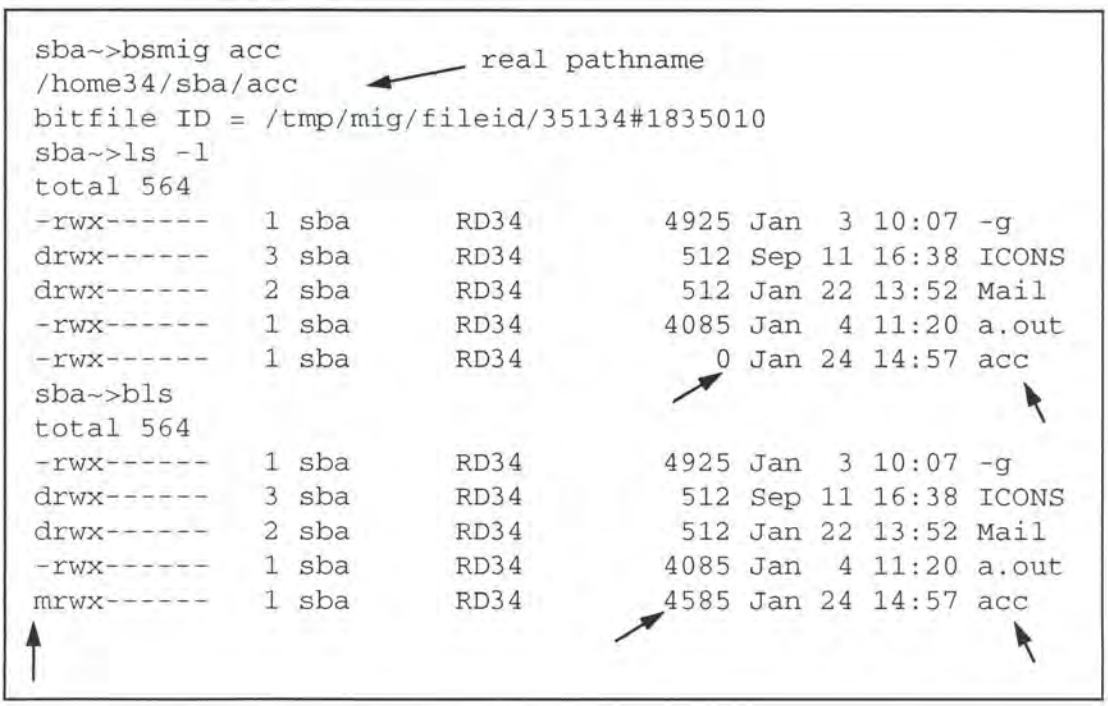


Figure 6.2 bsmig command.

6.3. Recall Functionality

In this part, we show a way to perform explicit recall functionality. Only explicit recall operation is implemented here.

bsrec command

bsrec filename.

Example:

bsrec /home34/sba/toto recalls the file */home34/sba/toto* from the bitfileID of */home34/sba/toto* containing the real filename (i.e. */tmp/mig/home34/sba/toto*) resides.

*bsrec se** , recalls all migrated files within the working directory whose file names start by *se*.

```
sba~>bsrec acc
Recalling /home34/sba/acc ...   Please wait ...
recall completed
sba~>ls -l
total 576
-rwx------   1 sba      RD34        4925 Jan  3 10:07 -g
drwx------   3 sba      RD34         512 Sep 11 16:38 ICONS
drwx------   2 sba      RD34         512 Jan 22 13:52 Mail
-rwx------   1 sba      RD34        4085 Jan  4 11:20 a.out
-rwx------   1 sba      RD34        4585 Jan 24 15:00 acc
sba~>bls
total 576
-rwx------   1 sba      RD34        4925 Jan  3 10:07 -g
drwx------   3 sba      RD34         512 Sep 11 16:38 ICONS
drwx------   2 sba      RD34         512 Jan 22 13:52 Mail
-rwx------   1 sba      RD34        4085 Jan  4 11:20 a.out
-rwx------   1 sba      RD34        4585 Jan 24 15:00 acc
```

Figure 6.3 bsrec command.

6.4. Intercepting file access request

In this last part, we show a way to intercept a file access request without modifying the kernel source code.

Here, we have chosen to intercept the open routine. As we have seen in chapter 4, a way to achieve this, is to generate a pseudo file system type by implementing a pseudo device driver.

Once the pseudo device driver is implemented, and suppose the name of the pseudo file system type is "zfs", the following steps to install it is described below.

1. A new file /etc/conf/mfssys.d/zfs that contains a line : zfs zfs_

2. A new file /etc/conf/sfsys.d/zfs that contains a line : zfs Y

3. Compile environment :

The compile environment can be found in /etc/conf/bin. A modified version of the script /etc/conf/bin/idcc for our own purpose is outlined below :

```
#------------------start of script idcc-----------------
#!/sbin/sh -

CC="/usr/bin/cc"
IDCC_FLAGS=`cat /etc/conf/bin/IDCC_FLAGS`
exec ${CC:-cc} -o SP_Driver.o $IDCC_FLAGS -WM,-G${KERNEL_GNUM:-16} -
W0 -Dunix -D__pyrsoft -c "$@"
#------------------end of script idcc----------------------
```

This script can be used to compile the zfs_driver.c and produce a SP_Driver.o. File /etc/conf/bin/IDCC_FLAGS contains all necessary compile-flags for the system's include files :

```
./idcc zfs_driver.c
```

4. Build-environment :

Copy the SP_Driver.o to /etc/conf/pack.d/zfs and rebuild a new Unix kernel with /etc/conf/bin/idbuild -S. This command will then configurate the zfs file system type and link zfs/SP_Driver.o to the Unix kernel. The new Unix will be located in /etc/conf/cf.d.

You can verify that zfs is included in the new unix by typing :

```
cd /etc/conf/cf.d
nm unix | fgrep zfs
```

5. Reboot-procedure :

It is recommended to remove /etc/.new_unix **after** successfully calling idbuild (this is an indication to the shutdown-scripts to install the new build unix instead of the original in the root-directory) and copy the new Unix from /etc/conf/cf.d/unix to /unix_zfs (or another name);

It is **very** important to have always access to the original /unix in order to reboot in case the Unix will run into panic.

Conclusion

The main objective of this work was to study how a file migration solution or a HSM solution can be integrated in HSMS/HSMS-CL product.

Our walk was, first, to try to understand the HSMS and HSMS-CL products. Secondly, we tried to understand the HSM concepts. Thirdly, we are interested to know how these concepts are applied in two existing HSM products which are DataMgr and EpochMigration. Then, we have tackled the feasibility analysis of file migration solution in a UNIX environment. But here, we are focused only on the HSM operations, particularly on implicit recall operation, because it is one of the most difficult problems to realize a HSM solution.

At the end, we have proposed a prototype which shows a way to intercept file access request in a SINIX system and to distinguish resident files from non-resident files. This prototype implements also two commands, one, which realizes an explicit migration operation and the other, which realizes an explicit recall operation.

It is important to note that migration does not replace data backup and archival. The purpose of migration is mainly to free space on higher performance, higher cost storage, whereas the purpose of backup and archival is to enable recovery of lost or inaccessible data and retrieval of point-in-time stored data. So a storage management solution should have the ability to backup files before migrated and to ensure the integrity of data stored on the storage server.

It would be intersting in further studies to see how objects (rather than only files) could be migrated or managed with HSM concepts. This case could be well applied in database fields. As databases are big size files, it would be interesting to be able migrate a segment (e.g. records) of a database, along with criteria, rather than the entire file to and from the storage server.

Glossary

API — See application programmer interface.

application programmer interface (API) — A set of calling conventions that defines how a service is invoked through software package.

Archival — Long-term saving of files that are no longer required. The files are deleted from the processing level once they have been backed up.

ARCHIVE — BS2000 software product which saves files and job variables logically. ARCHIVE has an internal interface with HSMS and implements the HSMS action statements.

archive — Management unit for files under HSMS management, consisting of the archive definition and the associated repository. HSMS makes a distinction between five archive types. There are archives concerning DMS files : backup archives, long-term archives, migration archives. The other archives are used to save workstation files (node-files), node backup archives, node long-term archives. Furthermore, HSMS distinguishes between private archives, which may be accessed by the archive owner only, and public archives, which are available to all users.

Backup — The periodic creation of copies of the data inventory to permit the restoration of data lost due to hardware errors or inadvertent deletion, etc. Can also be used to reorganize disk storage.

bitfile — The contents of a non-resident or shadowed file on a bitfile server. Bitfiles are never modified; however, if a shadowed file is subsequently modified or if a non-resident file is recalled and modified, then new bitfile and bitfile ID are created when the file is prestaged or migrated.

bitfile ID — Pointer to a particular bitfile on a bitfile server.

client — The system that establishes a network connection with another system (typically called the server) and that requests and receives action from the server.

DataMgr	One of the Advanced Archival Products, Inc. AMASS Storage management System family of products. DataMgr provides transparent file migration from magnetic disk to less costly storage, such as optical disk jukebox and tape libraries.
daemon	A background process, often perpetual, that performs system wide public function.
demand migration	Migration that is initiated by a daemon when space is exhausted during normal operation. See also file migration.
explicit migration	Migration that is initiated by a user request through a command. See also file migration.
explicit recall	Recall operation initiated by a user command. See also recall.
file migration	Process of moving selected files from file systems to slower and cheaper storage.
high watermark	The file system utilization percentage at which automatic (demand or periodic) migration starts.
HSM	Hierarchical Storage Management.
HSMS	Hierarchical Storage Management System : BS2000 software product offering such functions as migration, backup, archival, and data transfer, implemented in a storage hierarchy and in archives.
HSMS-CL	Client version of the HSMS software, running on UNIX workstations.
IEEE Mass Storage System Reference Model	A model that specifies the following functions : (1) the bitfile client; (2) the bitfile server; (3) the storage server; the bitfile mover; (5) the physical volume repository; and (6) the name server.
implicit recall	Automatic recall initiated by a daemon. See also recall.
logical size	The space that a resident file takes up, or that a non-resident would take up if it were not migrated.
low watermark	The file system utilization percentage at which automatic (demand or periodic) migration will stop migrating files and, if prestaging is in effect, when they will start prestaging files.
mass storage	The resources of computer system or computer network (e.g. magnetic/optical disks, magnetic tapes, magnetic cartridges) that provide long term storage for massive amounts of data.

migrated file See non-resident file.

migration See file migration.

migration policy A set of factors that are defined in configuration files to select which files are to migrate and when and where they are to be migrated.

Network File System A network service developed by Sun Microsystems that lets a (NFS)program that run on the same computer use data that is stored on a different computer on the same network as if it were on its own disk.

NFS See Network File system.

non-resident file A file that have been migrated. The file's contents have been moved to a bitfile server.

periodic migration migration that is initiated by an administrative job during non-peak times to bring space utilization down to a predifined level.

recall A process which retrieves a migrated file and copies it back to local disk.

wrappers A thin layer of code between the virtual filesystem and the unix filesystem at kernel level of the system. Wrappers determine if the current file is being accessed is a migrated file or not. If yes, manage the recall operation. In addition, wrappers allow to intercept application requiring more disk space available by migrating files.

Bibliography

[AMS94]
AMASS Archival Management Storage System, *Reference Manual, Version 2.2,* May 1994.

[AMS95]
AMASS Archival Management Storage System, *System Overview, Version 4.2* January 1995.

[DDK96]
Didier Derck, *Decentralized Repository for Data Backup Management,* Master Thesis, Institute of Computer Science, Facultés Universitaires Notre-Dame de la Paix, Namur, June 1996.

[DTM94]
DataMgr, *User's Guide, Version 2.2,* December 1994.

[DTM95]
DataMgr, *System Overview, Version 2.2*, January 1995.

[EPM92]
An Epoch Systems, *Technical Summary*, April 1992.

[FTS93]
Siemens Nixdorf Informationssysteme AG, *Programmer's Guide : Writing File System Types*, December 1993.

[GRA95]
John R. Graham, *Solaris 2.x Internals and Architectures,* J. Ranade Workstation Series, McGraw-Hill Inc., ISBN 0-07-911876-3. pp. 135-137, 1995.

[HEU93]
Gerald R. Heuring, *Pragmatics of File Migration,* Thesis, DCS, Urbana Champaign, University of Illinois, Report N° UICSDCS-R-93-1827.

[HSCL95]
Siemens Nixdorf Informationssysteme AG, *External Specification for HSMS-CL,* August 1995.

[HSMS 94]
Siemens Nixdorf Informationssysteme AG, *BS2000 as Backup Server in Open Universe. Company-wide Backup with HSMS V2.0.* Brief Description. July 1994.

[HSMS95]
Siemens Nixdorf Informationssysteme AG, *HSMS/HSMS-SV V2.0B. Hierarchical Storage Management System.* July 95.

[IEEE90]
Sam Coleman and Steve Miller, *Mass Storage Reference Model, Version 4,* IEEE Technical Committee on Mass Storage Systems and Technology.

[KLEI86]
Steven R. Kleiman, *Vnodes : An Architecture for Multiple File System Types,* in Sun Unix pp. 238-247, Proceedings of Summer Usenix Conference, Atlanta, GA, 1986.

[LIN94]
David S. Linthman, *Playing the Storage Shell Game,* Open Computing, August 1994.

[MIL91]
Ethan Miller, *File Migration on the Cray Y-MP at National Center for Atmospheric Research,* CSD, University of California, Berkeley.

[ROS90]
David S. H. Rosenthal, *Evolving The Vnode Interface*, Sun Microsystems, 1990.

[STE95]
Larry Stevens, *Hierarchical Storage Management,* Open Computing, May 1995.

Appendix

A1. Specification

Name : bsmig.c

```
Procedure f
purpose : create a bitfile
pre : s is pointer to a character
post : /

Procedure fattr
purpose : create a bitfile ID from a file name
pre : s is a pointer to a character
post : /

Procedure : copier
purpose : copy a file in /tmp/mig directory
pre : s is a pointer to a character
post : /

Procedure : IsExcluded
purpose : verify if  a given file excluded from migration or not
pre : s is a pointer to a character
post : if the file specified by s has an entry in the file
/home34/sba/excluded return 1, else return 0

Procedure : mig
purpose : migrate a given non empty file to /tmp/mig directory
pre : fname is a pointer to a character
post : /
```

Name : bsrec.c

```
Procedure  : copier
purpose : copy a given file into another
pre : s1, s2 are pointers to a character
post : /

Procedure : fattr
purpose : recall a given file from /tmp/mig directory
pre : s is pointer to a character
post : if the recall operation is succeeded then return 0, else return -1
```

Name : zfs_driver.c

procedure : zfs_init
purpose : initialize the vfs structure
pre : - vswp is a pointer to the vfssw structure declared in /usr/include/sys/vfs.h file
- fstype is an integer
post : /

Procedure : zfs_sync
purpose : to show only that zfs is a "true" file system type that is recongnize by the system.
pre : - vfsp is a pointer to the vfs structure declared in the /usr/include/sys/vfs.h file.
- flag is an integer
- cred is pointer to the cred structure declared in the /usr/include/sys/cred.h file.
post : /

Procedure : zfs_dummy
purpose : for unsupported operations call
pre : /
post : /

A2. Implementation

```
/*****************************************************************/
/*                          SNI, January 1996                    */
/*                                                               */
/*          Prototype which realizes migration functionality     */
/*                                                               */
/*                           by Souleymane BAH                   */
/*****************************************************************/


  #include <unistd.h>
  #include <sys/types.h>
  #include <sys/mkdev.h>
  #include <ftw.h>
  #include <dirent.h>
  #include <sys/stat.h>
  #include <fcntl.h>
  #include <libgen.h>
  #include <string.h>
  #include <stdio.h>
  #include <errno.h>

  #define TAILLE 512
  #define DMIG "/tmp/mig"

  const char *pattern;
  int user_func (const char *,const struct stat *,int,struct FTW*);

  void f (char *s)
  {
    int ret;
    char *wp;
    char dir[1024]="/tmp/mig";
    wp=s+1;
    for (;;)
    {
        if ((wp=strchr(wp,'/'))==NULL) break;
        else {
              (*wp)='\0';
              strcat(dir, s);
             /* printf("%s\n", dir);*/
              if ((ret=mkdir(dir,00700))!=0) perror(" ");
              strcpy(dir, "/tmp/mig");
              (*wp)='/';
              wp++;
            }
    }
  }

void fattr(char *s)
{
  FILE *fp;
  char dir[1024]="/tmp/mig/fileid/";
  char tab[1024], ti[1020], td[1024];
  char dir1[1024]=DMIG;
  struct stat buffer;
   int ret=-1;
```

```
   ret = lstat(s,&buffer);
   if (ret == 0) {
      sprintf(ti,"%d\n", buffer.st_ino);
      sprintf(td,"%d\n", buffer.st_dev);
   }
  strcat(dir1, s);
  strcat(tab, ti);
  tab[strlen(tab)-1]='\0';
  strcat(tab, "#");
  strcat(tab, td);
  strcat(dir, tab);
  printf("bitfile ID = %s\n", dir);
  fp=fopen(dir, "w");
  fputs(dir1, fp);
  fclose(fp);


}

void copier(char *s)
{
   char tamp[TAILLE];
   int lec, ecr, n;
   char tab[1024]=DMIG;
   struct stat buffer;

   strcat(tab, s);
   lstat(s,&buffer);
   if(s) {lec=open(s, 0);
   ecr=creat(tab, (buffer.st_mode&S_IAMB));
   while ((n=read(lec, tamp, TAILLE)) > 0)
     write(ecr, tamp, n);
     }

}

void mig (char *fname)
{
    char tab[1024]=DMIG;
    struct stat buffer;
    int fd;

    lstat(fname, &buffer);
    if ((getuid()==buffer.st_uid) || (buffer.st_uid==0)) {
    f(fname);
    strcat(tab, fname);
    if (access(tab, F_OK)==0) remove(tab);
    copier(fname);
    remove(fname);
    fd=creat(fname, (buffer.st_mode&S_IAMB));
    close(fd);
    fattr(fname);
    }
    else printf("you are not the file owner\n");


}
```

```
main (int argc,char *argv[])
{
    char *resolved_name[512];
    int returned=-1;
    char *rec, *mig;


    rec="/home34/sba/bsrec";
    mig="/home34/sba/bsmig";
    if (argc!=2) {
       printf("erreur sur le nombre d arguments !!\n");
       return 1;
    }

    pattern=argv[1];

  if (*pattern!='/') {
     realpath(pattern, resolved_name);
     strcpy(pattern, resolved_name);
  }
  if (strcmp(pattern, mig)==0 || strcmp(pattern, rec)==0)
    { printf("error !! This file is excluded from migration\n");
      return 1;
    }

    returned = nftw ("/home34/sba",user_func,FTW_PHYS);
    return 0;
}

int user_func (char *fn, struct stat *stat_ptr, int typ, struct FTW ftw_info)
{
    if ((stat_ptr -> st_mode &S_IFMT)==S_IFREG)
       if(gmatch(fn,pattern))
          {
              mig(fn);
/*            printf("match: %s \n",fn);*/
          }

  return 0;
}
```

```
/*************************************************************************/
/*                    SNI, January 1996                                  */
/*                                                                       */
/*            Prototype which realizes recall functionality              */
/*                                                                       */
/*                           by Souleymane BAH                           */
/*************************************************************************/


  #include <unistd.h>
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>
  #include <string.h>
  #include <stdio.h>
  #include <errno.h>
  #include <stdlib.h>
  #include <sys/param.h>

  #define TAILLE 512
  #define DMIG "/tmp/mig"

char *frec;

void copier(char *s1, char *s2)
{
   char tamp[TAILLE];
   int lec, ecr, n;
   struct stat buffer;

   lstat(s1,&buffer);
   if (s1) {lec=open(s1, 0);
   ecr=creat(s2, (buffer.st_mode&S_IAMB));
   while ((n=read(lec, tamp, TAILLE)) > 0)
     write(ecr, tamp, n);}

}


int fattr(char *s)
{
  FILE *fp;
  char dir[512]="/tmp/mig/fileid/";
  char tab[1024], ti[1024], td[1024], tamp1[1024];
  struct stat buffer;
  int maxl=1024;
   int ret=-1;
   printf("Please wait ...\n");
   ret = lstat(s,&buffer);
   if (ret == 0) {
      sprintf(ti,"%d\n", buffer.st_ino);
      sprintf(td,"%d\n", buffer.st_dev);
   }
  strcat(tab, ti);
  tab[strlen(tab)-1]='\0';

  strcat(tab, "#");
  strcat(tab, td);
  strcat(dir, tab);
  if (access(dir, F_OK)==0) {
```

```
      fp=fopen(dir, "r");
      fgets(tamp1, maxl, fp);
      printf("%s\n", s);
      remove(s);
      copier(tamp1, s);
      remove(dir);
      return 0;
    }
    else return -1;
}

main(int argc, char *argv[])
{ char *resolved_name[512];

    if (argc!=2) {
       printf("erreur sur le nombre d arguments !!\n");
       return 1;
    }

    if (*argv[1]!='/') {
       realpath(argv[1], resolved_name);
       strcpy(argv[1], resolved_name);
    }
    if (fattr(argv[1])==0) printf("recall completed\n");
      else printf("error!!!\n");
return 0;

}
```

```
/*****************************************************************/
/*                      SNI, January 1996                        */
/*                                                               */
/*         Prototype which allows to distinghish resident from   */
/*  non-resident file by generating a m bit for each migrated file */
/*                                                               */
/*                          by Souleymane BAH                    */
/*****************************************************************/


# include <sys/types.h>
# include <errno.h>
# include <sys/stat.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#define TLIGNE 512

#define DMIG "/tmp/mig"



main()
{
   struct stat buffer;
   int ret=-1;
   char tamp[TLIGNE];
   char tamp1[TLIGNE];
   char tampd[TLIGNE];
   char dir[1024]=DMIG;
   char dir1[TLIGNE];
   int maxl=1024;
   FILE *fp;
   FILE *fdp, *fmp;

   char *c, c1;
   int i,j,k,l;
   char aux[512], aux1[512];
   char aux2[512]="/home34/sba/";

   system("ls -l /home34/sba > titi");
   fp=fopen("/home34/sba/titi", "r");

    c=fgets(tamp, maxl, fp);
   while (c!=NULL) {

       for (j=39; tamp[j]!='\0'; j--);
       for (i=0; tamp[54+i]!='\0'; i++)
           aux[i]=tamp[54+i];
           aux[i-1]='\0';
     if (tamp[0]=='-') {
             system("pwd > wd");
             fdp=fopen("/home34/sba/wd","r");
             if (fgets(tampd, maxl,fdp)!=NULL)
             fclose(fdp);
             j=strlen(tampd)-1;
             tampd[j]='/';
             tampd[j+1]='\0';
             strcat(tampd,aux);
```

```
            strcat(dir,tampd);
            strcat(aux2,aux);
              if (access(dir, F_OK)==0) {
                ret = lstat(aux2,&buffer);
                 if (ret == 0)
                  if ((buffer.st_size)==0)
                     tamp[0]='m';


  system("ls -l /tmp/mig/home34/sba > lili");
  fmp=fopen("/home34/sba/lili", "r");
  c1=fgets(tamp1, maxl, fmp);
  while (c1!=NULL) {
    if (tamp1[0]=='-') {
      for (i=0; tamp1[54+i]!='\0'; i++)
         aux1[i]=tamp1[54+i];
      aux1[i-1]='\0';
      if(strcmp(aux, aux1)==0) {
        for (i=32; i<40; i++) tamp[i]=tamp1[i];
        break;
      }
      c1=fgets(tamp1, maxl, fmp);
    }
    else  c1=fgets(tamp1, maxl, fmp);
  }
  }
            aux2[0]='\0';
            printf ("%s", tamp);
            strcpy(dir,DMIG);
            c=fgets(tamp, maxl, fp);
     }
     else {
            aux2[0]='\0';
            printf ("%s", tamp);
            c=fgets(tamp, maxl, fp);
          }
  }

  fclose(fp);

   return 0;
}
```

```
/*****************************************************************/
/*                     January 1996                              */
/*                                                               */
/*      Prototype which allows to intercept file access request  */
/*                      (See zfs_open routine)                   */
/*                                                               */
/*                       by Souleymane BAH                       */
/*****************************************************************/

/*------------------start of the file zfs_driver.c--------------*/
#include <stdio.h>

#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/resource.h>
#include <sys/signal.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/buf.h>
#include <sys/vfs.h>
#include <sys/vnode.h>
#include <sys/proc.h>
#include <sys/disp.h>
#include <sys/file.h>
#include <sys/fcntl.h>
#include <sys/flock.h>
#include <sys/kmem.h>
#include <sys/uio.h>
#include <sys/conf.h>
#include <sys/mman.h>
#include <sys/pathname.h>
#include <sys/debug.h>
#include <sys/vmmeter.h>
#include <sys/cmn_err.h>

#include <sys/fs/ufs_fs.h>
#include <sys/fs/ufs_inode.h>
#include <sys/fs/ufs_fsdir.h>
#ifdef QUOTA
#include <sys/fs/ufs_quota.h>
#endif
#include <sys/dirent.h>       /* must be AFTER <sys/fs/fsdir.h>! */
#include <sys/errno.h>
#include <sys/sysinfo.h>

#include <vm/hat.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_map.h>
#include <vm/seg_vn.h>
#include <vm/rm.h>
#include <sys/swap.h>

#include "fs/fs_subr.h"
```

```
/*#include <sys/fs/specfifo.h>      /* this defines PIPE_BUF for ufs_getattr()
*/
extern int fs_nosys();

/*
 * zfs vnode operations
 */
extern  int zfs_open();
extern  int zfs_close();
extern  int zfs_read();
extern  int zfs_write();
extern  int zfs_ioctl();
extern  int zfs_getattr();
extern  int zfs_setattr();
extern  int zfs_access();
extern  int zfs_lookup();
extern  int zfs_create();
extern  int zfs_remove();
extern  int zfs_link();
extern  int zfs_rename();
extern  int zfs_mkdir();
extern  int zfs_rmdir();
extern  int zfs_readdir();
extern  int zfs_symlink();
extern  int zfs_readlink();
extern  int zfs_fsync();
extern  void zfs_inactive();
extern  int zfs_fid();
extern  void zfs_rwlock();
extern  void zfs_rwunlock();
extern  int zfs_seek();
extern  int zfs_frlock();
extern  int zfs_space();
extern  int zfs_getpage();
extern  int zfs_putpage();
extern  int zfs_map();
extern  int zfs_addmap();
extern  int zfs_delmap();
extern  int zfs_allocstore();

/*
 * zfs vfs operations:
 */
extern int zfs_dummy();
extern int zfs_sync();

struct vfsops zfs_vfsops = {
      zfs_dummy,  /* mount */
      zfs_dummy,  /* umount */
      zfs_dummy,  /* root */
      zfs_dummy,  /* statvfs */
      zfs_sync,   /* sync */
      zfs_dummy,  /* vget */
      zfs_dummy,  /* mountroot */
      zfs_dummy,  /* swapvp */
      fs_nosys,
      fs_nosys,
      fs_nosys,
      fs_nosys,
      fs_nosys,
```

```
        fs_nosys,
        fs_nosys,
        fs_nosys,

};


/*
 * zfs vnode operations
 */

struct vnodeops zfs_vnodeops = {
          zfs_open,
          zfs_close,
          zfs_read,
          zfs_write,
          zfs_ioctl,
        fs_setfl,
          zfs_getattr,
          zfs_setattr,
          zfs_access,
          zfs_lookup,
          zfs_create,
          zfs_remove,
          zfs_link,
          zfs_rename,
          zfs_mkdir,
          zfs_rmdir,
          zfs_readdir,
          zfs_symlink,
          zfs_readlink,
          zfs_fsync,
          zfs_inactive,
          zfs_fid,
          zfs_rwlock,
          zfs_rwunlock,
          zfs_seek,
        fs_cmp,
          zfs_frlock,
          zfs_space,
        fs_nosys,   /* realvp */
          zfs_getpage,
          zfs_putpage,
          zfs_map,
          zfs_addmap,
          zfs_delmap,
        fs_poll,
        fs_nosys,   /* dump */
        fs_pathconf,
          zfs_allocstore,
        fs_nosys,   /* filler */
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
```

```
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
        fs_nosys,
};


int     zfsfstype;
int     zfs_dbg = 0;


/*
 * Initialize the vfs structure
 */

void
zfs_init(vswp, fstype)
        struct vfssw *vswp;
        int fstype;
{
     struct inode *ip;
     int i;

     extern struct inode *ufs_inode; /* begin of ufs-inode table */
     extern struct inode *inodeNINODE; /* end of ufs-inode table */
     extern int ufs_ninode;          /* number of configured ufs-inodes */

      zfs_dbg = 0xdeadbeef;   /* to indicate: zfs_init() has been activated */
      /*
       * Associate vfs operations
       */

      vswp->vsw_vfsops = &zfs_vfsops;
      zfsfstype = fstype;
      printf("=======> ZFS: hello world; zfsfstype=%d\n", zfsfstype);

      /*
       * manipulation of incore ufs-inodes here:
       */

     ip = ufs_inode;
```

```
        printf("ZFS ==> Device number: %u\n", ip->i_dev);
        printf("ZFS ==> Inode Number: %u\n", ip->i_number);
        printf("ZFS ==> ufs_ninodes: %d\n", ufs_ninode);

        ip->i_vnode.v_op = &zfs_vnodeops;
        for (i = ufs_ninode; --i > 0; ) {
                ++ip;
                ip->i_vnode.v_op = &zfs_vnodeops; /* <-- manipulation itself */
        }


        return;
}

zfs_sync( vfsp, flag, cr)
struct vfs  *vfsp;
int         flag;
struct cred *cr;
{
        /*
         *  this routine is only to show you that zfs is a "true" fs type
         *  that is recognized by the system;
         *  set zfs_dbg to 1 (with ikdb); after typing a sync-command the
         *  following message appears on the console:
         */

           if (zfs_dbg == 1) {
              printf("zfs: sync\n");
           }
}

int
zfs_dummy()
{
        printf("zfs: dummy, unsupported operation called\n");
        return EINVAL;
}


int
zfs_open(vpp, mode, cr)
        struct vnode **vpp;
        int mode;
        struct cred *cr;
{
        int error;
        printf("===> Hello ZFS ! you are in open wrapper routine\n");
        error=ufs_open(vpp, mode, cr);
        return (error);
}

int
zfs_close(vp, flag, cnt, off, cr)
        struct vnode *vp;
        int flag;
        int cnt;
        off_t off;
        struct cred *cr;
{
        int error;
```

```
        error=ufs_close(vp, flag, cnt, off, cr);
        return(error);
}

int
zfs_read(vp, uiop, ioflag, cr)
          struct vnode *vp;
          struct uio *uiop;
          int ioflag;
          struct cred *cr;
{
         int error;

         error = ufs_read(vp, uiop, ioflag, cr);
           return(error);
}

int
zfs_write(vp, uiop, ioflag, cr)
          struct vnode *vp;
          struct uio *uiop;
          int ioflag;
          struct cred *cr;
{
         int error;

           error = ufs_write(vp, uiop, ioflag, cr);
           return(error);
}

int
zfs_ioctl(vp, cmd, arg, flag, cr, rvalp)
        struct vnode *vp;
        int cmd;
        int arg;
        int flag;
        struct cred *cr;
        int *rvalp;
{
        int error;
        error=ufs_ioctl(vp, cmd, arg, flag, cr, rvalp);
        return(error);
}

int
zfs_getattr(vp, vap, flags, cr)
        struct vnode *vp;
        struct vattr *vap;
        int flags;
        struct cred *cr;

{
        int error;
        error=ufs_getattr(vp, vap, flags, cr);
        return(error);
}

int
zfs_setattr(vp, vap, flags, cr)
        struct vnode *vp;
```

```
        struct vattr *vap;
        int flags;
        struct cred *cr;

{
        int error;
        error=ufs_setattr(vp, vap, flags, cr);
        return(error);
}


int
zfs_access(vp, mode, flags, cr)
        struct vnode *vp;
        int mode;
        int flags;
        struct cred *cr;

{
        int error;
        error=ufs_access(vp, mode, flags, cr);
        return(error);
}

int
zfs_lookup(dvp, nm, vpp, pnp, flags, rdir, cr)
        struct vnode *dvp;
        char *nm;
        struct vnode **vpp;
        struct pathname *pnp;
        int flags;
        struct vnode *rdir;
        struct cred *cr;

{
        int error;
        error=ufs_lookup(dvp, nm, vpp, pnp, flags, rdir, cr);
        return(error);
}

int
zfs_create(vp, nm, vap, excl, mode, vpp, cr)
        struct vnode *vp;
        char *nm;
        struct vattr *vap;
        enum vcexcl excl;
        int mode;
        struct vnode **vpp;
        struct cred *cr;

{
        int error;
        error=ufs_create(vp, nm, vap, excl, mode, vpp, cr);
        return(error);
}

int
zfs_remove(vp, nm, cr)
        struct vnode *vp;
        char *nm;
```

```
        struct cred *cr;

{
        int error;
        error=ufs_remove(vp, nm, cr);
        return(error);
}

int
zfs_link(tdvp, svp, tnm, cr)
        struct vnode *tdvp;
        struct vnode *svp;
        char *tnm;
        struct cred *cr;

{
        int error;
        error=ufs_link(tdvp, svp, tnm, cr);
        return(error);
}

int
zfs_rename(sdvp, snm, tdvp, tnm, cr)
        struct vnode *sdvp;
        char *snm;
        struct vnode *tdvp;
        char *tnm;
        struct cred *cr;

{
        int error;
        error=ufs_rename(sdvp, snm, tdvp, tnm, cr);
        return(error);
}

int
zfs_mkdir(dvp, dirname, vap, vpp, cr)
        struct vnode *dvp;
        char *dirname;
        struct vattr *vap;
        struct vnode **vpp;
        struct cred *cr;

{
        int error;
        error=ufs_mkdir(dvp, dirname, vap, vpp, cr);
        return(error);
}

int
zfs_rmdir(vp, nm, cdir, cr)
        struct vnode *vp;
        char *nm;
        struct vnode *cdir;
        struct cred *cr;

{
        int error;
        error=ufs_rmdir(vp, nm, cdir, cr);
        return(error);
```

```
}

int
zfs_readdir(vp, uiop, cr, eofp)
     struct vnode *vp;
     struct uio *uiop;
     struct cred *cr;
     int *eofp;

{
     int error;
     error=ufs_readdir(vp, uiop, cr, eofp);
     return(error);
}

int
zfs_symlink(dvp, linkname, vap, target, cr)
     struct vnode *dvp;
     char *linkname;
     struct vattr *vap;
     char *target;
     struct cred *cr;

{
     int error;
     error=ufs_symlink(dvp, linkname, vap, target, cr);
     return(error);
}

int
zfs_readlink(vp, uiop, cr)
     struct vnode *vp;
     struct uio *uiop;
     struct cred *cr;

{
     int error;
     error=ufs_readlink(vp, uiop, cr);
     return(error);
}

int
zfs_fsync(vp, cr)
     struct vnode *vp;
     struct cred *cr;

{
     int error;
     error=ufs_fsync(vp, cr);
     return(error);
}

void
zfs_inactive(vp, cr)
     struct vnode *vp;
     struct cred *cr;

{
     ufs_inactive(vp, cr);
}
```

```
int
zfs_fid(vp, fidpp)
     struct vnode *vp;
     struct fid **fidpp;

{
     int error;
     error=ufs_fid(vp, fidpp);
     return(error);
}

void
zfs_rwlock(vp)
     struct vnode *vp;

{
     ufs_rwlock(vp);

}

void
zfs_rwunlock(vp)
     struct vnode *vp;

{
     ufs_rwunlock(vp);

}

int
zfs_seek(vp, ooff, noffp)
     struct vnode *vp;
     off_t ooff;
     off_t *noffp;

{

     int error;
     error=ufs_seek(vp, ooff, noffp);
     return(error);

}
int

zfs_frlock(vp, cmd, bfp, flag, offset, cr)
     struct vnode *vp;
     int cmd;
     struct frlock *bfp;
     int flag;
     off_t offset;
     struct cred *cr;

{
     int error;
     error=ufs_frlock(vp, cmd, bfp, flag, offset, cr);
     return(error);

}

int
```

```
zfs_space(vp, cmd, bfp, flag, offset, cr)
     struct vnode *vp;
     int cmd;
     struct frlock *bfp;
     int flag;
     off_t offset;
     struct cred *cr;

{
     int error;
     error=ufs_space(vp, cmd, bfp, flag, offset, cr);
     return(error);

}



int
zfs_getpage(vp, off, len, protp, pl, plsz, seg, addr, rw, cr)
     struct vnode *vp;
     u_int off;
     u_int len;
     u_int *protp;
     struct page *pl[];
     u_int plsz;
     struct seg *seg;
     addr_t addr;
     enum seg_rw rw;
     struct cred *cr;

{
     int error;
     error=ufs_getpage(vp, off, len, protp, pl, plsz, seg, addr, rw, cr);
     return(error);
}

int
zfs_putpage(vp, off, len, flags, cr)
     struct vnode *vp;
     u_int off, len;
     int flags;
     struct cred *cr;

{
     int error;
     error=ufs_putpage(vp, off, len, flags, cr);
     return(error);

}

int
zfs_map(vp, off, as, addrp, len, prot, maxprot, flags, cr)
     struct vnode *vp;
     u_int off;
     struct as *as;
     caddr_t *addrp;
     u_int len;
     u_int prot;
     u_int maxprot;
     u_int flags;
```

```
        struct cred *cr;

{
        int error;
        error=ufs_map(vp, off, as, addrp, len, prot, maxprot, flags, cr);
        return(error);

}

int
zfs_addmap(vp, off, as, addr, len, prot, maxprot, flags, cr)
        struct vnode *vp;
        u_int off;
        struct as *as;
        caddr_t addr;
        u_int len;
        u_int prot;
        u_int maxprot;
        u_int flags;
        struct cred *cr;

{
        int error;
        error=ufs_addmap(vp, off, as, addr, len, prot, maxprot, flags, cr);
        return(error);

}

int
zfs_delmap(vp, off, as, addr, len, prot, maxprot, flags, cr)
        struct vnode *vp;
        u_int off;
        struct as *as;
        caddr_t addr;
        u_int len;
        u_int prot;
        u_int maxprot;
        u_int flags;
        struct cred *cr;

{
        int error;
        error=ufs_delmap(vp, off, as, addr, len, prot, maxprot, flags, cr);
        return(error);

}

int
zfs_allocstore(vp, off, len, cr)
         register struct vnode *vp;
         u_int off, len;
         struct cred *cr;
{
  int error;
  error=ufs_allocstore (vp, off, len, cr);
  return error;

}

/*---------------end of file zfs_driver.c--------------------*/
```