# BUMP
# The BRL/USNA Migration Project

*Michael John Muuss, BRL*

*Terry Slattery, USNA*

*Donald F. Merritt, BRL*

The US Army Ballistic Research Laboratory
The US Naval Academy

*ABSTRACT*

On UNIX® systems with many users, or systems that run very large problems, disk space management can be particularly difficult. Space management has generally been accomplished by scripts and programs for determining "disk hogs". Users have been expected to explicitly move their working files to some offline storage media, often using manual procedures and record keeping. In addition, UNIX programs attempting to write to a full file systems get an ENOSPC write error: "No space on device". Frequently, this behavior is not acceptable, especially where programs may execute for a long time.

This paper reports on the implementation of a solution to both these aspects of the disk space management problem, by providing a transparent file migration facility. The result of this software is UNIX filesystems that have the appearance of significantly more capacity than then underlying disk drives, freeing the user community from worrying about managing offline storage media.

The system administrator or a **cron** script may run a utility to cause certain files to be migrated to one of several levels of offline storage. The inode for each migrated file remains present in the filesystem, with special "file handle" data used to recover the file on subsequent access. When a migrated file is opened, the kernel will block the user process, wait for a special user-mode migration daemon to recover the file from backing storage, and then allow the user process to continue. This mechanism is entirely transparent to the user, except for the delay. When a process attempts to write onto a full file system, the system can be configured to block the process, start a "space management" migration function to create file system space, then resume the blocked processes.

The details of the kernel modifications, support daemons, and related software necessary to provide fully transparent file migration will be presented. In addition, the software described is Public Domain, Distribution Unlimited. Several vendors have already expressed plans to incorporate it into their products.

13 May 1995

# BUMP
# The BRL/USNA Migration Project


*Michael John Muuss, BRL*

*Terry Slattery, USNA*

*Donald F. Merritt, BRL*

The US Army Ballistic Research Laboratory
The US Naval Academy


## 1. BACKGROUND

Computer systems running UNIX range from personal computers to supercomputers. On systems with many users, or systems that run very large problems, disk space management can be particularly difficult. Historicly, UNIX space management has been accomplished by scripts and programs for determining ‘‘disk hogs’’ and mechanisms for users to explicitly move their files to some offline storage media (magnetic tape, removable disks, etc.), often using manual procedures and manual record keeping.

These traditional techniques have suffered from a number of serious pitfalls, some of which are not at all obvious. The most commonly heard complaint is that users are unclear about how to recover files that have been moved offline, what tape that files are on, how long the tape copies of relocated files will be kept, etc. Vigorous system administrators are often tempted to sweep their filesystems for files that have not been accessed recently, and then move these files to tape. Users are typically notified of this preemptive relocation of their files with a brief note, and typically no serious harm is done. However, if users have been working on a large software project for a long time, and they are using **make** to generate their binary files, the filesystem sweep may determine that many source files are not being used, because they have not even been accessed for a long time. If those files are preemptively moved offline, **make** will explode with error messages the next time it is run, because **make** uses **stat**(2) to build the dependency tree. This is but one example of files that have not been ‘‘accessed’’ in a long time that are still being actively ‘‘used’’.

UNIX programs attempting to write to a full file systems get an ENOSPC write error: ‘‘No space on device’’. Frequently, this behavior is not acceptable, especially in the supercomputer realm where programs may execute for a long time, after which there may not be enough space to write all the results to disk. Generally, it seems that most users would prefer that the system simply waited until additional space became available, perhaps to the accompanyment of warning messages, rather than aborting work in progress. If the out of space condition is recognized by the users, users can often open another window, or use another terminal, or influence a conveniently located colleague, to free up some additional space. It is most unfortunate that work in the process of being written to disk has always been lost in these circumstances.

This paper describes the implementation of a solution to both aspects of the disk space management problem. The software provides transparent file migration and archiving without requiring major changes to the UNIX kernel or filesystem. One result is UNIX filesystems that can have the appearance of significantly more capacity than then underlying disk drives, freeing the user community from worrying about managing offline storage media.

## 2. GOALS

There were two primary goals of this project: First, to create a file migration system for UNIX that provides filesystems that give the appearance of having significantly more online storage than the actual device that contains the filesystem. Second, all unmodified UNIX programs that do not examine the raw filesystem must not be able to detect any difference between regular files, and migrated files, except for possible delay in completing an open() operation on a migrated file.

Secondary goals were to achieve these features:

1) Have separation between migration *policy*, and migration *mechanism*, so sites that may need to modify the policy will not have to alter the migration mechanism.

2) Make minimal modifications to the UNIX kernel. The basic kernel routines should be installed like a device driver, and all interface ‘‘hooks’’ should be short, and conditionally compiled via #ifdef MIGRATION. Most of the essential functionality should be located in user-mode code.

3) No changes to the size or structure of the on-disk inodes. This permits this software to be installed on machines with existing filesystems, without needing to dump and reload all user files. This also minimizes changes to existing kernel code, dump/restore code, and standalone utilities.

4) Provide support for an arbitrary number and variety of secondary storage devices and recording methods. By using a highly structured and modular interface to the software that handles the secondary storage devices, recipients of this software will be able to easily support additional hardware, and adapt to novel devices, without having to change the fundamental mechanism of this system.

5) Provide extremely robust operation in the face of both system crashes and heavy system use. In essence, this software says ‘‘trust me with your files;’’ that trust must not be violated. Filesystem reliability and availability should be comparable to that of a UNIX system that is not running this software.

6) To have the capability of having multiple copies of migrated files located on secondary storage, for reliability,

7) To have the capability for leaving a copy of a migrated file online, so that either rapid in-migration or rapid space reclamation can be accomplished, depending on which resource is required first, access or storage.

8) To provide support for several types of secondary storage, and for staging files from one form of secondary storage to another. For example, small files might initially be migrated to some type of robotic mass storage, while larger files might go directly to operator mounted magnetic tape. Files on the robotic mass storage might be staged out to magnetic tape if they are not accessed within a few weeks.

Some consciously chosen limitations to this system are:

1) Only regular files can be migrated. It is not possible to migrate directories or special files.

2) To provide migration service only to the machine hosting the disk system. This software is not attempting to provide a CTSS-style ‘‘common filesystem’’ across multiple machines.

3) There is no relief for the problem of creating files that are larger than the online capacity of a single filesystem.

## 3. OVERVIEW

BUMP is a collection of user level tools, supported by a small set of kernel modifications, to provide the user and system administrator with facilities that allow files to be migrated to backing storage and then transparently restored when accessed. These tools allow the user or system administrator to identify files to migrate, force these files into a ‘‘pre-migrated’’ state, copy the pre-migrated files to backing storage, and release the disk storage associated with these files. A specially modified version of the standard **ls**(1) program will defeat the transparency feature of the migration system, to allow the user to identify files that have been migrated. Additional tools allow the user to recover files in the background for future use, and determine the amount of space taken by files in the migration system. The system administrator has tools available to coalesce sparsely populated migration volumes and to move migrated files between different levels in the storage hierarchy.

To migrate a set of files, the names of the selected files are collected in a migration-list file (see Figure 1, "Functional Diagram"), with an optional hint about future usage and optional comment regarding the reason for migration (e.g. sysadmin forced, etc). All files on this list are migrated to a special ‘‘migrate’’ directory that exists for each filesystem upon which BUMP has been configured to run. Files that have been migrated to the on-disk directory are in a ‘‘pre-migrated’’ state in which the online disk storage has not been released, but the original inode has been changed to mode IFMIG and an entry in the file database has been created. The archiver utility is run to make at least one copy (and typically two copies) of each pre-migrated file onto backing storage media. After the copy has been made, the file is marked as ‘‘dual-migrated’’, because both the online copy and the secondary storage copies exist. From the dual-migrated state, the file can be instantly returned to full online status if the migrated inode is opened, or the on-disk pre-migrated copy may be unlinked to free disk storage if space runs low.

When a migrated file is accessed, either by an attempt to **open**(2) it, or by an attempt to **exec**(2) it, the kernel recognizes the migrated inode type, blocks the process attempting the action, and sends a message to a user level daemon requesting that the affected file be reloaded. The daemon runs the archiver utility to copy the file back into the ‘‘migrate’’ directory on the filesystem where the reload is to occur, and once restored to the dual-migrated state, ‘‘unmigrates’’ it. The kernel then learns of the reload completion from the daemon and allows the waiting process to continue.

Free disk space management is automatically provided by a system-wide ‘‘low space’’ disk usage threshold. When the threshold is crossed the kernel notifies the daemon to begin reclaiming free space. The kernel also informs the daemon when file space is completely exhausted, blocking processes that attempt to write on that file system until the daemon creates free space and sends a reply back to the kernel.

The system’s operation is easily tailored for specific sites, both in terms of the selection policy for the files to migrate and the type of hardware used for archiving. A hierarchy of storage levels is supported for sites with more than one type of archival media. Policy decisions about which and how many files to migrate are easily adjustable by the system administrators, making it easy to adapt to varying requirements at different sites.

## 4. FILE FINDING TOOLS

The first step in the migration process is to identify candidate files to migrate, perhaps using a policy of selecting files with the largest size or size*age product being selected first. Because selecting files to migrate is independent of the actual archiving mechanism, each site may implement its own selection policy. A combination of the **migsweep** and UNIX **find**(1) utilities can aid system administrators in selecting files based on either the size*age product or some other policy. The only restriction on which files may be migrated is that they be regular files (i.e. type IFREG).

Additionally, the **migsweep** tool will permit each user to designate (in a ‘‘.precious’’ file) the names of files that should never be migrated, up to a certain amount of ‘‘permanent’’ disk storage allocated for that user. In this way, a user can be certain that the ‘‘.profile’’ file and other small files that are very frequently used will not be migrated. Otherwise, getting logged in could become very time consuming!

Users will be provided with another tool to voluntarily migrate files that they know will not be needed for some period of time. The migrate list produced by these tools may include optional fields for a comment and hints about possible future reload times to be used by the archiving mechanism in a multi-level storage hierarchy.

## 5. OUT MIGRATION

The migration file list, kept in a disk file to survive system crashes and reboots, is read by the **migout** tool, that in turn migrates each file to a special per-filesystem ’migration’ directory. See Figure 2, "Out Migration (migout)". In this process, migout creates a database entry for each migrated file, allocates the pre-migrated inode in the migration directory, and calls the mig_makemigrated() system call. mig_makemigrated() moves the block pointers from the original file’s inode to the pre-migrated inode in the migration directory, zeros the original inode block pointers, stores the file’s ‘‘handle’’ in the now empty block pointer area, and changes the original file’s type to IFMIG. Files in this state are termed ‘‘pre-migrated’’ because they have been migrated from the normal UNIX filesystem into the special on-disk migration directory, but have not been copied to any other storage media. This operation does not free any storage on the affected filesystem, and in fact, uses another inode for each pre-migrated file. However, it is an atomic operation that results in the file’s data blocks being allocated to an inode in a protected directory where advisory file locking may be used to guarantee the integrity of files during archiving.

The ‘‘file handle’’ is a unique sequence number given to each file in the system that allows easy location of all copies of an migrated file, regardless of the storage methods used. File handles are used as the index into the file database to find all occurrences of a migrated file in the archiving system. This indirection is needed to allow files to be moved from one volume to another or from one storage level to another, without having to hunt down and modify the migrated inode to reflect a change. A file handle is composed of a 32-bit source host identifier and a 32-bit file identifier. Including the source host identifier in the migrated file handle prevents problems from arising when filesystems are accidentally used on the wrong computer system. This could easily happen when removable-media filesystems are present in a site, or it could also happen if a set of dump tapes were reloaded onto a machine other than the originating machine. If the file handle consisted of only a sequence number, and a filesystem from another system was mounted, this would have two undesirable consequences. First, it would grant the requesting user access to protected files owned by other users, and second, having reloaded the migrated files, it would remove the copy of the file from secondary storage, preventing the legitimate owner of the file from reloading it later.

## 6. ARCHIVING (STAGING)

Once the file has been pre-migrated, it must be copied onto at least one backing storage method in order to free the on-line disk blocks. See Figure 3, "Migration Archiving (migarch)." The archiving (or staging) process is provided by **migarch**, a utility that copies files from one storage method to another, including both to and from the on-disk migration directory where pre-migrated files reside. See Figure 4, "Migration Archiving, FROM disk", and Figure 5, "Migration Archiving, TO disk". Migarch will typically be instructed to create at least two copies of each migrated file to facilitate recovery of data written to media that may be subsequently damaged.

A storage method is defined as a type of media (e.g. tape) and a recording format (e.g. ANSI labels). The input to migarch is a list containing the filehandle, destination method, and number of copies for each file to be copied. The file database is searched to find all possible sources of this file, the ‘‘closest’’ copy is determined, an operator request issued for the source and destination volumes to be mounted, the data copied, and the file database updated. This process is repeated for each copy of each file. Destination tape volumes always start out being empty. Migarch is used to coalesce partial volumes, copying files from several partial volumes onto one empty volume. By always writing onto a previously empty tape volume, the system helps guarantee that an existing volume will never be corrupted by unintentional overwriting (such as by power loss during a write operation).

Migarch also allows groups of files to be copied to a single set of volumes, with the selection based on a combination of database parameters such as owner, group, or size. In this case, the input list is built by searching the file database (or the filesystem) for all files matching the desired criteria. One possible use of this feature would be to cause each volume to contain files owned by a single user, for security reasons.

Pre-migrated files that have been archived to backing storage may either remain in the file system or may be removed to free the associated disk space. If they remain in the filesystem, they are called ‘‘dual-migrated’’ files. In this state, one of two operations may occur: 1) a reload request arrives, causing the migin tool to quickly convert the file back to its normal state (see below), or 2) the filesystem runs short of space and the online copy of the dual-migrated file is unlinked to reclaim the disk space it consumed.

## 7. MODIFICATIONS TO THE UNIX FILESYSTEM

In order to implement migrated files, it was necessary to have some way to distinguish between regular files, and migrated files. The most obvious way to achieve this would have been to use another bit in the inode that indicates that the inode is migrated. While the Berkeley 4.2 BSD and 4.3 BSD filesystems have additional space in the on-disk inodes that such a bit could be placed, the current System V filesystems do not have any additional space. However, in both kinds of UNIX systems there are several unused combinations of the IFMT file type bits in the i_mode inode field. Therefore, a single one of these unused combinations is given the symbolic name IFMIG, and is used to mark inodes that are migrated regular files. It is this lack of an additional bit that prevents migrating directory inodes and ‘‘special file’’ inodes, as well as regular files.

When an inode is of type IFMIG, representing a migrated file, it is necessary to store the eight byte ‘‘file handle’’ in the on-disk inode. In the Berkeley 4.2 BSD and 4.3 BSD filesystems there are 16 bytes marked ‘‘reserved, currently unused’’ in the ic_spare[] field, so on the Berkeley implementation the first eight bytes out of the 16 spare bytes are used to store the file handle. The System V on-disk inode has no unused space, so eight bytes of the disk block number array are reused (overloaded) to store the file handle information when the inode is of type IFMIG. This location is referred to as i_fhandle.

There are a number of cases where additional features could have been provided if space existed to store the file handle in all inodes, such as the Berkeley inode format allows. For example, this would have allowed files to have been reloaded for reading only, and then subsequently deleted from the disk, without having to write a new copy to secondary storage. This would also have allowed the concept of a ‘‘dual-migrated’’ file to have been handled in a somewhat simpler manner. A future effort will be to modify the System V filesystem to have the larger inode space required, and then to provide these additional features.

As a result of these file system modifications, it is necessary to update all system utilities that read the raw filesystem. Most notable among these are **fsck**, **dump**, and **restore**.

## 8. THE DEFINITION OF KERNEL OPERATIONS AND THE MESSAGE PROTOCOL

This section describes the communications protocol used between the UNIX kernel mig.c module, and the user-mode migration daemon. For illustrative purposes, interfaces to existing kernel code are drawn from the work done interfacing BUMP to Cray UNICOS 3.0.10 on an XMP. Only the additions are shown, to prevent disclosing any proprietary software. Similar additions exist for 4.2 BSD and 4.3 BSD kernels.

There are two conceptual ‘‘layers’’ to the protocol that is used between the kernel and the migration daemon. The lower level is the basic message-passing mechanism by which the kernel and the user mode daemon exchange chunks of data (messages). The upper level is the definition of the structure of the messages, the meaning of the various message types, and the nature of any expected actions or responses.

### 8.1. THE MESSAGE PASSING MECHANISM

To establish the communication path, the user mode daemon must **open**(2) the special kernel interface device, /dev/mig0. /dev/mig0 will ordinarily be owned by root, and mode 0600, to protect the interface from unauthorized use.

When the daemon opens the /dev/mig0 interface, it receives a normal UNIX file descriptor as the return value from the **open**() sys-call. This file descriptor is used for all subsequent communications. The /dev/mig0 ‘‘driver’’ code has special checks to ensure exclusive use of the interface, ie, only one user mode process may have the Kernel/Daemon message interface open at any one time.

Once the daemon has the interface open, communication between the user mode daemon and the kernel is via normal UNIX **read**() and **write**() system calls.

The daemon’s UNIX **write**() system call is vectored through the cdevsw table to the kernel routine migwrite(). The daemon may send a message to the kernel at any time. The driver code has been arranged so the kernel always has one local message buffer to read a message into. Therefore, the kernel will always be able to accept and process a message from the daemon. The byte count argument to the **write**() system call must be exactly the size of one message buffer, ie, sizeof(struct mig_msg), or an EIO error will be returned. A direct consequence of this is that the daemon must perform one **write**() system call for each message sent to the kernel. There is no message ‘‘batching’’ mechanism.

The daemon’s UNIX **read**() system call is vectored through the cdevsw table to the kernel routine migread(). The byte count argument to the **read**() system call must be at least as large as the size of one message buffer, so that one entire message can be sent to the daemon in a single operation. Partial reads are not permitted, to simplify the kernel code, and to prevent the daemon from losing track of the start-of-message-buffer location in the byte stream comming from the driver. If the kernel has one or more messages waiting for the daemon, the **read**() system call returns exactly one message back to the daemon, without delay. If the kernel has no messages waiting for the daemon, then the daemon is blocked at interruptible priority until a message arrives.

If the daemon wishes to ‘‘sense’’ the presense of a message, with an optional wait-for-message delay, the UNIX select() call may be used, that vectors through the cdevsw table to the kernel routine migselect(). The migselect() routine indicates that there is read capacity, ie, a message is ready to be read, when that is the case. Attempts to sense the write capacity of the device always return a ‘‘true’’ indication.

When a close() sys-call is performed on the file descriptor returned from the open for /dev/mig0, the kernel routine migclose() will be called. At the time the message passing interface is closed, special cleanup action is taken by the kernel to deal with any messages that the daemon had left outstanding.

Should the the daemon die unexpectedly, or perform an exit() sys-call before closing the file descriptor to the /dev/mig0 interface, the normal UNIX kernel code that cleanly closes all open file descriptors before destroying the hulk of the dead process will ensure that a suitable call of migclose() will occur, even though the daemon process did not explicitly perform one. This will ensure that the exclusive use semaphore (kernel variable mig_daemon_is_open) is properly cleared, so that when the daemon is restarted, continued operation will be possible.

The contents, organization, and semantics of the message contents are the domain of the higher level.

### 8.2. THE DEFINITION OF A MESSAGE

The format of the data exchanged between the kernel and the daemon is defined by the C structure ‘‘mig_msg’’, defined in kernel header file migration.h. At present, it looks like this:

```
struct mig_msg {
        int             ms_magic;       /* MIG_MSG_MAGIC */
        int             ms_id;          /* ID of msg, for kernel */
        int             ms_op;          /* operation, see below */
        int             ms_result;      /* may contain errno */
        dev_t           ms_dev;                 /* relevant device */
        struct fhandle       ms_handle;     /* file handle */
        struct mig_inode_id ms_inode;      /* for mig_iunmigrate() */
} mc_msg;
```

The field ms_magic must always be set to the value MIG_MSG_MAGIC, or the message is discarded as ‘‘noise’’, and an error is logged. The field ms_id is a unique message ID that is issued by the kernel. The kernel will never have more than one message outstanding to the daemon with the same message ID. The daemon is required to echo this ID number back to the kernel in any reply message that might be sent. The field ms_op defines the operation, or purpose, of a message. The remaining fields, ms_result, ms_dev, ms_handle, and ms_inode contain valid values only when so noted in the documentation for a specific value of ms_op. Note that the contents of the ms_inode structure are to be considered ‘‘opaque’’ by the daemon, and are intended to be passed intact as one of the parameters to the mig_iunmigrate() sys-call. The daemon should never store or perform any operations on the the ms_inode element.

There are two forms of messages that the kernel can send: blocking messages and asynchronous messages. Blocking messages require a response from the daemon, and asynchronous messages do not require a response. It is important to note that the term ‘‘blocking’’ does not imply that the daemon must answer a message immediately, nor does it imply that the daemon may not answer other messages first. The term ‘‘blocking’’ signifies that there is a user mode process that has been blocked, awaiting the response message from the daemon, and that a kernel message structure remains committed to this transaction until the daemon replies.

There are only two types of messages that the daemon may send, and both are issued in response to a ‘‘blocking’’ kernel message: MIG_D2K_DONE messages, and MIG_D2K_FAIL messages. This simplicity of response messages was intended to simplify the kernel’s job when ‘‘inventing’’ proper responses to messages that were outstanding when the daemon closes the /dev/mig0 interface.

Conceptually, either the kernel or the user mode daemon may transmit a message to the other at any time. Messages from the daemon to the kernel will always be processed immediately. Because the generation of kernel to daemon messages happens in the context of some process other than the daemon process, the message is stored in a message structure, and queued for processing by the daemon. This queueing mechanism requires that there be a pool of message structures, and at present, this pool is of fixed size (kernel parameter MIG_NMSG). When the entire pool of message structures has been committed to use, any additional requests will block waiting for a structure to become available. Therefore, it is desirable to set

MIG_NMSG large enough to handle the anticipated number of transactions during ‘‘high demand’’ periods.

### 8.3. USER-SETTABLE MODES

There are three new ‘‘mode bits’’ on every process that can be read and set by the user, using the mig_getflag() and mig_setflag() sys-calls.

The first bit, called SPACERETRY, determines kernel behavior when a user executes a **write**() sys-call to a file on a full filesystem. When SPACERETRY=0, the traditional UNIX behavior occurs, and the **write**() sys-call returns -1, with variable ‘‘errno’’ set to ENOSPC. When SPACERETRY=1, the **write**() sys-call vectors to mig_nospace(), that will block the user’s process until additional space becomes available, and then allows the **write**() operation to transparently proceed.

The remaining two bits, called MIGNOTRANSP and MIGCANCEL, control the way the kernel handles an attempt to open a migrated file. When MIGNOTRANSP=0, the user wants transparent access to a migrated file. The process will block until the file has been reloaded, after which the **open**() will complete normally. If the user process receives a signal, perhaps SIGINT (eg, ‘‘^C’’) when the user gets impatient, the in-migration operation will be permitted to proceed, if MIGCANCEL=0. If MIGCANCEL=1, the daemon will be notified to abort the in-migration operation.

When MIGNOTRANSP=1, the user does not want transparent file access, instead preferring to have the **open**() return -1 immediately with variable ‘‘errno’’ set to EMIGRATED. If MIGCANCEL=0, the daemon is notified to initiate an asynchronous reload, in the background. If MIGCANCEL=1, no communication with the daemon occurs.

### 8.4. RELOADING MIGRATED FILES

When a user process attempts to open a migrated file, the **open**() system call is vectored to the kernel routine **open**(), that calls copen(). copen() contains a block of code to open an existing file, which is supplemented with the following block of code in the Cray os/sys2.c module:

```
#ifdef MIGRATION
        if( !u->u_error &&
            (ip->i_mode & IFMT) == IFMIG &&
            !(mode&FTRUNC) )  {
                extern struct inode *mig_reload();

                /* Note replacement of "ip" after reload */
                if( (ip = mig_reload( ip )) == (struct inode *)0 )
                        return;
        }
#endif
```

where copen() calls the mig_reload() subroutine. On the Cray, similar code is added to the gethead() routine in os/exec.c.

The mig_reload() routine checks the status of the MIGTRANSP and MIGCANCEL mode bits on this process to determine the precise handling of the operation. If MIGNOTRANSP=0 (transparent operation), the daemon is sent a message with ms_op=MIG_K2D_RELOAD_BLOCK, with ms_handle and ms_inode having the file handle and device/inode-number information pertaining to the migrated inode. This message is sent to the daemon using the mig_daemon_wait() routine, which will block waiting for a reply message from the daemon. If the daemon reply message is MIG_D2K_DONE, then the **open**() proceeds normally. If the daemon reply message is MIG_D2K_FAIL, then the **open**() fails, returning -1, and the value in the ms_result field of the daemon reply message is used as the value of ‘‘errno’’. The daemon should do all database operations for this message based strictly on the value of the ms_handle element.

If a signal is received while the process is blocked in mig_daemon_wait(), a notification message is sent to the daemon, the signal is posted to the user process, and the **open**() will return -1, with the value of ‘‘errno’’ being set to EMIGFAIL. The message sent to the daemon will have the same values of ms_handle and ms_inode as the original MIG_K2D_RELOAD_BLOCK message had. If MIGCANCEL=0, then the message sent to the daemon is MIG_K2D_SILENCE_RELOAD, which informs the daemon to proceed with an outstanding RELOAD_BLOCK operation, but not to send any further notification back to the kernel, ie, treat the operation as if it had originally been of type MIG_K2D_RELOAD_ASYNC. If MIGCANCEL=1, then the message sent to the daemon is MIG_K2D_CANCEL_RELOAD, which informs the daemon that the reload of this file is no longer required, and if possible, it should be aborted. If the daemon has already "committed" to the reload operation, no harm is done by allowing the reload to complete.

In this protocol, there is the a potential for a non-harmful race condition between the kernel queueing a SILENCE or CANCEL message to the daemon just before the daemon begins sending a DONE or FAIL message to the kernel for that operation. Therefore, for best results the daemon should always check to see if there are any additional kernel-to-daemon messages waiting, before the daemon sends reply messages to the kernel. When the race condition exists, the daemon will send a reply message to the kernel that is no longer expected. This will be detected in the kernel routine mig_process_response(), and in this case, the kernel will simply note the occurrence of the race by sending the daemon a MIG_K2D_UNEXPECTED message for adding to the daemon log files. In this message, the original message is duplicated for return, with ms_id being the new message ID ms_result being set to the ms_id field of the unexpected message just received, and ms_op being set to MIG_K2D_UNEXPECTED. If the kernel is unable to obtain a message buffer to log this error condition, then a console printf() is performed, with the message "WARNING: mig_process_response: unexpected daemon reply, id=%d". This behavior was necessary on the out of buffers condition to prevent a potential buffer deadlock condition.

If MIGNOTRANSP=1 (non-transparent operation), the **open**() fails with "errno" set to EMIGRATED. If MIGCANCEL=0, then the daemon is sent a MIG_K2D_RELOAD_ASYNC message, with ms_handle and ms_inode having the pertinent information. This permits the daemon to initiate an asynchronous reload operation for the file. The daemon should do all database operations for this message based strictly on the value of the ms_handle element. It is anticipated that the daemon would probably queue these asynchronous requests at a lower priority level than requests where there is a user process actively waiting a reload operation. If MIGCANCEL=1, then no daemon notification happens at all.

If the daemon is not running when mig_reload() is called, the **open**() fails, and "errno" is set to EMIGOFF. No access to a migrated inode is permitted while the daemon is not running.

### 8.5. LOW SPACE AND NO SPACE

There is a system-wide disk usage threshold mig_minfreefrags that can be read by anyone using the mig_getflag() sys-call, and can be set by the superuser with the mig_setflag() sys-call.

It is planned that in a future version of this software, where tighter integration with existing system utilities would be possible, that the minimum space threshold would be settable on a per-filesystem basis. On Berkeley UNIX systems, this would most likely become part of the in-core mount table information, set by an extra field in /etc/fstab.

When a filesystem transitions from an amount of free storage above the threshold to an amount below the threshold, the routine mig_lowspace is called. Here is the code fragment from the Cray fs/c1/c1alloc.c routine:

```
#ifdef MIGRATION
{
        extern int mig_minfreefrags;

        if( fp->s_tfree >= mig_minfreefrags &&
            (fp->s_tfree - reqblks) < mig_minfreefrags )
               mig_lowspace(dev, fp->s_tfree);
}
#endif
```

The mig_lowspace() routine sends a message to the daemon, with ms_op set to MIG_K2D_LOWSPACE, ms_dev being the device running low on space, and ms_result set to the amount of storage remaining. No daemon response to the kernel is expected.

On receipt of the LOWSPACE message, the daemon has the option (depending on configuration parameters) of initiating a Space Management function to make additional space on the device. This might include removal of some pre-migrated files, and/or the initiation of an immediate filesystem sweep and out-migration operation, depending on the site-specific configuration.

Note that the daemon has to be prepared for getting several LOWSPACE messages concerning the same device within a short period of time, as the storage level oscillates around the threshold level. These should be collapsed into a single event within the daemon.

When an attempt is made to allocate blocks on a filesystem that is full, the routine mig_nospace() is called. Here is the code fragment from Cray fs/c1/c1alloc:

```
#ifdef MIGRATION
        if( mig_nospace( dev ) == 0 )
                goto retry;
#endif
        prdev("ERROR: alloc.c: no available free space",dev);
```

mig_nospace() first checks the setting of the process SPACERETRY mode bit. If SPACERETRY=0, the kernel calls mig_lowspace() with a space level of zero to note the condition, the error ENOSPC is returned from mig_nospace(), and allocation fails.

If SPACERETRY=1, then a message is sent to the daemon with ms_op=MIG_K2D_NOSPACE and ms_dev set to the relevant major/minor device code. The kernel then blocks the user process until a reply message is received from the daemon. If the response is MIG_D2K_DONE, then the storage allocation is retried. There is no race condition here, because if another process has consumed the newly made storage before this process retries the allocation, mig_nospace() will be called again, and the operation repeats. If the response from the daemon is MIG_D2K_FAIL, then the allocation operation is abandoned, and an ENOSPC error is returned to the user process. It is not anticipated that the daemon would ever return a MIG_D2K_FAIL code to a NOSPACE message, as that defeats the purpose of this feature.

If the user process fields a signal while it is blocked waiting for the daemon reply to the NOSPACE message, then the kernel sends the daemon a message with ms_op=MIG_K2D_CANCEL_NOSPACE and ms_dev set to the major/minor device code. Note that the daemon is still expected to act on the no space condition, even though the user process is no longer blocked waiting on space, ie, treat the message as a NOSPACE message with 0 blocks left. Perhaps a better name would have been SILENCE_NOSPACE. There is a potential race condition here, as with the MIG_K2D_RELOAD_BLOCK message described earlier; it is not harmful, and mig_process_response() will take the same remedial action.

Note that the daemon should be prepared for multiple processes to encounter a NOSPACE condition on a given device within a fairly short time of each other. The daemon is responsible for holding all of them, and not releasing them until it is known that some storage becomes available, which the daemon can learn about from (a) the Space Management task on that device completing, and (b) periodic sampling of free space levels.

If mig_nospace() is called and the user has set SPACERETRY=1, but the migration daemon is not running, rather than returning an error, the kernel adopts a simple retry strategy, to prevent the user process from seeing unwanted ENOSPC errors. If the user has no signals pending, a non-interruptible kernel sleep will be initiated in mig_nospace(), with a one minute timeout. This will cause an allocation retry once a minute, until additional storage becomes available, or the process receives a signal, perhaps SIGINT (eg, ‘‘^C’’) when the user gets impatient. This polling behavior is not optimal, especially if several dozen processes need additional space, but it prevents the very desirable SPACERETRY feature from evaporating when the daemon is not running.

Thanks to this feature, processes that have SPACERETRY=1 should never see an ENOSPC error return. This is extremely valuable for long-running processes that write all their answers to disk just before exiting.

### 8.6. TRUNCATING MIGRATED FILES

Whenever a process attempts to truncate a migrated file to zero length, the routine mig_trunc() is called. Truncations to non-zero lengths cause a normal mig_reload() operation, as described above. The code fragment from Cray fs/c1/c1iget.c. Also, if it is desired to take ‘‘core dumps’’ on top of migrated files, a similar modification will be required in os/sig.c

```
#ifdef MIGRATION
        /* Reload the file prior to truncation, if new size > 0 */
        if((ip->i_mode & IFMT) == IFMIG)  {
                if( size == 0 )
                        u->u_error = mig_trunc( ip );
                else
                        u->u_error = mig_reload( ip );
                if( u->u_error )
                        return;
        }
#endif
```

When a migrated file is truncated to length zero, the kernel sends a message to the daemon with ms_op=MIG_K2D_TRUNCATE and ms_handle set to the appropriate file handle, after which, the kernel converts the inode back into a regular file (i_mode=IFREG) with length zero, and the normal truncate operation proceeds.

Typically, for migrated files whose data resides on some form of backing store, the daemon would move the database entries for the backing store copies into an ‘‘age, then queue for volume reclamation’’ queue, for subsequent Volume Management operations.

If the daemon is not running when mig_trunc() is called, the truncate operation fails, and ‘‘errno’’ is set to EMIGOFF. The contents of migrated inodes may not be altered while the daemon is not running, to prevent the migrated file database from becoming inconsistent with the state of the filesystem.

### 8.7. UNLINKING MIGRATED FILES

Whenever a process attempts unlink a migrated file, the routine mig_unlink() is called. The code fragment from Cray os/sys4.c, routine unlink():

```
#ifdef MIGRATION
        /*
         *  If this is the very last link to a migrated file,
         *  inform the migration system, and allow it the opportunity
         *  to note (and perhaps refuse) the operation, before
         *  removing the directory entry or dereferencing the inode.
         */
        if( ip->i_nlink == 1 && (ip->i_mode & IFMT) == IFMIG )  {
                if( (u->u_error = mig_unlink( ip )) != 0 )
                        goto out;
        }
#endif
```

When the last link to a migrated inode is removed, the kernel sends a message to the daemon with ms_op=MIG_K2D_UNLINK and ms_handle set to the appropriate file handle, after which, the kernel converts the inode back into a regular file (i_mode=IFREG) with zero size, and the normal unlink operation proceeds.

Typically, for migrated files whose data resides on some form of backing store, the daemon would move the database entries for the backing store copies into an ‘‘age, then queue for volume reclamation’’ queue, for subsequent Volume Management operations.

If the daemon is not running when mig_unlink() is called, the unlink operation fails, and ‘‘errno’’ is set to EMIGOFF. Migrated inodes may not be removed while the daemon is not running, to prevent the migrated file database from becoming inconsistent with the state of the filesystem.

**8.8. THE SYSTEM CALL INTERFACE**

In addition to the message passing interface, the kernel support for the migration system also provides several additional system calls.

In the present implementation, it was decided that these system calls would not be coded using the normal kernel sysent[] table, but would be handled by a private mechanism, so as to minimize the amount of existing kernel code that would have to be altered, and to prevent having to make vendor-specific system call interface modules for inclusion /lib/libc.a, the C runtime library. When a vendor installs this code in their system, they would presumably assign system call numbers, and add the interfaces to the C runtime library, for slightly increased speed and clarity.

To the applications programmer, the new migration system calls are indistinguishable from direct system calls. In the remainder of this paragraph, the details of the present implementation will be discussed, and then the difference will be ignored henceforth. All use of these new system calls is expected to be via the library routines provided in libmig.a, which establishes contact with the new system call interface in the kernel by opening device /dev/mig1, that will ordinarily be mode 0666 to permit general use. The interface routines in libmig will bundle up the system call number and arguments into a buffer, and **write**() it to /dev/mig1. Kernel routine migwrite() will copy this buffer into kernel space, and call the routine indicated by the system call number. The interface into the migration system call routines is identical to the interface seen when calling via the sysent[] table, to permit easy conversion to the direct method.

**8.9. FLAG MANIPULATION**

```
mig_getflag( cmd, pid )
int     cmd;
int     pid;

mig_setflag( cmd, pid, value )
int     cmd;
int     pid;
int     value;
```

There are presently four kernel parameters that may be read or altered using these two system calls. The SPACERETRY, MIGNOTRANSP, and MIGCANCEL parameters are one-bit quantities, and are stored on a per-process basis. The THRESHOLD low space threshold is presently a single integer value that applies to all filesystems (see LOWSPACE remarks, above). The ‘‘pid’’ argument must specify a live process, to avoid an ESRCH error. A ‘‘pid’’ value of zero is interpreted to indicate the pid of the process performing the system call. If the process specified by ‘‘pid’’ belongs to a different user, and the process performing the system call is not running as the superuser, then an EPERM error is returned.

When cmd=MIG_FLAG_SPACERETRY, the one bit parameter SPACERETRY is read or written. With a value of 0, writing to a full filesystem returns an ENOSPC error, while with a value of 1, writing to a full filesystem will always succeed, although potentially with some delay until free storage is available.

When cmd=MIG_FLAG_NOTRANSP, the one bit parameter MIGNOTRANSP is read or written. When cmd=MIG_FLAG_CANCEL, the one bit parameter MIGCANCEL is read or written. See the earlier remarks on reloading migrated files for a detailed description of the effects of these bits.

When cmd=MIG_FLAG_THRESHOLD, a mig_getflag() will return the current threshold upon which the daemon will be notified of a low space condition. Only the superuser may alter this parameter with mig_setflag(); non-privileged use will result in an EPERM error. Only non-negative values are permitted. Note that in this case the value of ‘‘pid’’ has no significance, but must be valid.

The SPACERETRY, MIGNOTRANSP, and MIGCANCEL bits are carried in the process structure p_flag word. They are inherited across forks, so that all child processes run with the same storage and migration behavior as the parent. Typically, a user would set the desired operating mode of his shell, and then all processes will behave in the desired manner. In order to permit these bits to be inherited by child processes, a one-line change needs to be applied to the fork() routine. In the subroutine newproc() in Cray os/fork.c:

```
#ifdef MIGRATION
        rpp->p_flag |= (rip->p_flag & (SRTIM|SCPUS|
                        SSPACERETRY|SMIGCANCEL|SMIGNOTRANSP));
#else
        rpp->p_flag |= (rip->p_flag & (SRTIM|SCPUS));
#endif
```

**8.10. CREATING A MIGRATED INODE**

```
mig_makemigrated( source, dest, handle )
char    *source;
char    *dest;
struct fhandle *handle;
```

This system call is intended for use by the out-migration tool. Only processes running as superuser may use this system call; other users will get an EPERM error. In ordinary use, the ‘‘source’’ file will be some user file, and the ‘‘dest’’ file will be in the migration directory for that filesystem.

The file named by ‘‘source’’ must already exist and be a regular file (i_mode=IFREG), and the file named by ‘‘dest’’ must not yet exist. Both must be on the same filesystem. The destination file is created in mode 0400, and is given the same size as the source file. Then, all of the disk blocks are transferred from the source file to the destination file, and removed from the source file. Finally, the source file is changed from file type regular (i_mode=IFREG) to migrated (i_mode=IFMIG), and the file handle given in ‘‘handle’’ is stored in an implementation-specific location within the source file’s inode. Note that the source file retains it’s original ownership, access modes, size indicator (i_size), access and modification times. However, it now no longer is using any storage for disk blocks. Any attempt to access this file will result in notification of the migration daemon, as described in the previous section.

**8.11. UNMIGRATING AN INODE**

```
mig_unmigrate( source, dest )
char    *source;
char    *dest;

mig_iunmigrate( source, idest )
char    *source;
struct mig_inode_id *dest;
```

Two system calls exist for reversing the effect of the mig_makemigrated() system call. The mig_unmigrate() form takes two path names, and is intended for human-driven diagnostic and disaster-recovery uses, and is not used by the production migration software. The mig_iunmigrate() form uses an opaque object of type ‘‘mig_inode_id’’ such as is found in the ms_inode field of kernel to migration daemon messages like MIG_K2D_RELOAD_BLOCK.

In ordinary use, the ‘‘source’’ file will be in the migration directory, and the ‘‘dest’’ file will be the corresponding migrated user file. The ‘‘source’’ file must be a regular file (i_mode=IFREG), and the ‘‘dest’’ file must be a migrated file (i_mode=IFMIG). Both files must be the same size. The ‘‘dest’’ file is converted back into a regular file, and then the disk blocks are moved from the ‘‘source’’ to the ‘‘dest’’ file. At this point, only the ‘‘change’’ time on the ‘‘dest’’ file will have been affected by the mig_makemigrated() and mig_iunmigrate() process; all other inode fields will be exactly as they were before the inode was migrated. Finally, the ‘‘source’’ file is unlinked.

**8.12. FILE STATUS**

One of the goals of this project was to implement a file migration capability that was so transparent that, except for additional delays for moving files in from backing storage, there would be no user visible differences. One implication of this is that the stat() system call must not indicate that a file is migrated --

otherwise, every application program that looks at the i_mode field of the inode (eg, find(1), du(1), etc) would have to be modified to know about the new inode type, IFMIG. That would not have been transparent at all! Instead, the stat() system call has been modified so that migrated inodes seem to be of regular file type, IFREG. This required the following change to Cray os/sys3.c routine stat1():

```
#ifdef MIGRATION
        /*
         *  Migrated files look like regular files to all users.
         *  Programs that care about the difference should use
         *  the mig_stat() sys-call instead.
         */
        if( (ip->i_mode & IFMT) == IFMIG )
                ds.st_mode = (ip->i_mode & ~IFMT) | IFREG;
#endif
```

Having made this modification, this raises the question of how a program that desired to know the true status of an inode can obtain it. This lead to the creation of two additional system calls:

```
mig_stat( name, statp, handle )
char            *name;
struct stat     *statp;
struct fhandle      *handle;

mig_lstat( name, statp, handle )
char            *name;
struct stat     *statp;
struct fhandle      *handle;
```

mig_stat() functions exactly like stat(). For non-migrated files, the file handle structure contains all zeros, while for migrated files, the file handle structure is non-zero, and contains the appropriate migration information copied from the inode. Note that in the later case, the file type will still be IFREG, not IFMIG, so that code that may be handed the stat structure would not have to be concerned with the extra file type.

For kernels that support the Berkeley concept of a symbolic link, the mig_lstat() subroutine is to the lstat() system call, as mig_stat() is to the stat() system call.

### 8.13. ERROR CODES

The existing set of kernel error codes that system calls can return in ‘‘errno’’ have been supplemented by several error codes that are specific to support for the migration system.

errno=EMIGRATED is returned when a user attempts to access a file that has been migrated, and the user has requested non-transparent access. This error is the result of the system honoring the request for non-transparency, and does not signify any difficulty.

errno=EMIGFAIL is returned when a wait for a transparent migration operation is interrupted by a signal.

errno=EMIGOFF is returned when a user attempts to access or delete a file that has been migrated, and the migration daemon is not running. No accesses to migrated files are permitted until the migration daemon has been restarted, so that the migration database remains consistent with the filesystem. The user should contact an operator or system administrator to have the daemon restarted.

errno=EMIGNLOC is returned on an attempt to migrate a file between two different filesystems, or when the source file is not local to the executing machine, eg, is an NFS file on a remote server. This error only occurs in the mig_makemigrated() and mig_unmigrate() system calls, and should only be seen by superuser processes.

## 9. THE MIGRATION DAEMON

Consistent with the principles of good modular operating system design, and in order to keep the required kernel additions to a minimum, most of the real work to handle file reload operations and out of space conditions is delegated by the kernel to the user-mode migration daemon process. In turn, the migration daemon itself does very little more than prioritize and queue requests from the kernel, and spawn various other processes to execute the needed migration tools. In particular, the daemon can spawn multiple copies of the in-migration tools, and it can also initiate a space management procedure (typically a shell script) for every filesystem that is running low on available disk storage.

When the migration software is installed on a machine, the migration daemon becomes an integral part of the operating system software on that machine. The migration daemon plays the same kind of critical role with the UNIX kernel filesystem functionality as the Internet “super-server” **/etc/inetd** and the domain name server **named** play for the UNIX network functionality. In ordinary operation, the migration daemon is not expected to die. However, if the daemon should die (or be killed), the kernel will make reasonable responses to all filesystem requests that are made. In particular, if a filesystem runs out of space when the daemon is not running, SPACERETRY=1 operation for reliable file writing is still provided, using a simple, less efficient all-kernel technique. If the migration daemon is not running, users will be prohibited from opening or deleting migrated files, although operations that affect only the inode will still be permitted, such as **chmod**(2), **rename**(2) or **mv**(1). This behavior is necessary to keep the migration system databases synchronized with the state of the filesystem. If the migration daemon is not running, the system should be considered to be experiencing a serious problem. Fortunately, it should always be possible for the superuser to log in on the console to take remedial action. This implies that crucial system files such as **/bin/sh** should not ever be migrated. It would be wise if the main system directories **/bin**, **/lib**, **/etc**, and **/usr/bin** were always exempted from out-migration. It would be better still if the entire root and **/usr** filesystems were never subjected to out-migration. Not only will this keep the system more responsive at a very small penalty in online storage used, but it will also ensure that all files needed for effecting system repairs will be available online when such repairs are called for.

## 10. IN MIGRATION

Reload requests are sent from the kernel to the daemon using the protocol described earlier. The migration daemon will fork and start a **migin** process to cause the file to be reloaded. See Figure 6, "In-Migration Function (migin)". **migin** runs the **migarch** utility to copy the file back into the file system if it is not already in place (i.e. dual-migrated). When the file is reloaded into the appropriate file system's migration directory, the daemon performs a mig_iunmigrate() system call, and notifies the kernel of the successful reload. If **migarch** experiences unrecoverable errors while trying to read every one of the multiple copies of the migrated file, then errors are logged in the migration log file, and an appropriate error is return through the daemon to the user process.

Files that are online in the filesystem have no existence in the secondary storage, because of the inability to store a file handle in the inode of a regular (i_mode IFREG) file. Therefore, when a file has been reloaded, all of the copies on secondary storage should be considered obsolete. However, to provide disaster recovery, the secondary storage copies of the migrated file can not immediately be turned over to Volume Management for reclamation. Instead, the secondary storage copies must be aged for a minimum of twice the backup interval before the storage can be reclaimed. If a file is removed, and then the filesystem is reloaded, it will still be available for a few additional days. Thus, any file that has been reloaded will be marked as obsolete in the file database, but will continue to be available for disaster recovery until the volume on which it is stored is reclaimed.

## 11. SPACE MANAGEMENT

To enable fully automatic recovery from disk space shortages, the migration system utilizes two kernel-to-daemon messages. The LOWSPACE message is sent when the freespace on a filesystem falls below the configured threshold and the NOSPACE message is sent when a filesystem is full. Upon receiving either of these messages, the daemon starts the **migspace** utility, that implements a system-specific policy for creating additional space. See Figure 7, "Space Management (migspace)". Typically, this policy would be to first consult the migration database to see if the affected file system has any dual-migrated files whose online disk storage can be immediately reclaimed. If this does not provide a sufficient amount of space, a list of new migration candidates should be built, and an out-migration operation would be initiated to move these files to backing storage. The worst case can occur when a few large files fill an entire filesystem, requiring all other files to be migrated to secondary storage.

Conservative sites, or sites that do not have 24-hour operator coverage may choose to configure the **migspace** script to create space only by unlinking all dual-migrated files, and then to give up and wait for human intervention. In this case, processes needing additional file space will be blocked until additional space becomes available, or some human kills them.

Note that some care needs to be taken to ensure that **migout** processing gets priority access to the tape drive in a single tape drive system.

## 12. DATABASES

There are several databases and intermediate files used by the BUMP tools; 1) a file database to map file handles into file location, 2) a volume database to identify the location and type of a storage volume, 3) a list of files to be staged from one archiving method to another, containing source file handle, number of copies to make, etc. All databases and intermediate files use the same basic format and are manipulated by a common set of routines. The format is an ASCII text file, with each record newline terminated and each field terminated with a vertical bar ‘‘|’’ character. Database fields that may require updating occupy a preset number of characters, so that the field may be updated in place. Concurrent update is prevented by using the appropriate kernel file locking features.

Using a simple file format for the databases which can be manipulated by the standard UNIX text processing tools results in a significant economy. Everything from simple enquiries up to the most sophisticated queries may be resolved using simple combinations of **grep**, **awk**, **ed**, and the rest of the UNIX text processing tools. Creating management reports on storage utilization can be handled with small Shell scripts that are easily tailored to the specific needs of individual sites. Using a text file format also permits ordinary text editors to be used to examine and modify the databases during development. It is anticipated that this convenience will prove similarly useful when disaster recovery is required.

Performance of this simple strategy is not anticipated to be a factor. If each database record requires 100 bytes, then the information for 10,000 migrated files will use a single megabyte of storage for the database. A large system may expect to have several hundred thousand migrated files at any time, totaling perhaps hundreds of gigabytes of storage, yet the migration database will remain comparatively modest in size.

## 13. PHILOSOPHY & IMPLICATIONS

The illusion of having unlimited on-line disk storage can be a great convenience for users. However, files may be migrated to automatic devices with recall times that can be measured in fractions of minutes, and files that have been migrated to devices that require operator intervention, such as conventional magnetic tape, will typically require several minutes per recall. The trade-off between the convenience of having extra storage and additional delay is certain to have uneven user appeal. Assuming that the file migration system has been well implemented, the success or failure of the file migration system in a particular environment will depend on having a migration policy that suits the needs of the most important users. This is why there has been such a strong emphasis placed on separating the migration *policy* from the migration *mechanism* — because no single migration policy will be able to meet the needs of all sites.

The most challenging environment to implement a successful migration policy in is unfortunately the environment that UNIX usually excels at: the highly interactive timesharing environment. Balancing the need for high interactivity with the need to have significantly more online storage than the actual capacity of the underlying filesystem will be difficult. Success is likely only if (a) the users perceive the benefits of additional convenient file storage as outweighing the inconvenience of an occasional delay in file access, and (b) the file migration policy has been tuned so that an average users ‘‘working set’’ of files is not ordinarily selected for migration, so that reload delays are incurred infrequently.

It is important to distinguish between the functions of the file migration system, the filesystem backup system, and the private user tape handling facility (sometimes also called an ‘‘archiving system’’, a usage that conflicts with the usage in this paper). Some operating systems, such as Cray's COS operating system, attempt to integrate parts of all three functions into the general filesystem capability. It is the purpose of this project only to provide a file migration capability, and not to disturb or significantly alter existing backup and user tape handling conventions. The file migration system is intended to provide the appearance of UNIX filesystems that are significantly larger than the actual capacity of the disk hardware. The filesystem backup system is intended to provide disaster recovery, so that a failed disk drive can be restored to the same state that it had at the time the last backup tapes were written. The backup system is also often used to help recover from serious user errors, where files are inadvertently deleted. When the deleted files have not been modified since the last backup tapes were written, then the files can be restored from the backup tapes, and the user error is undone. Finally, the user tape handling facility is intended to allow users to selectively but permanently save files that may be needed again in the future, but will not be needed online for a significant amount of time.

To prevent filesystem backups from taking an absurd amount of tape, and an even more absurd amount of time, it is necessary to modify the backup procedures to discriminate between regular files and migrated files. For migrated files, the backup procedure should record only the inode information onto the dump tapes, and not the actual contents of the file. This is consistent with the definition of the backup system as providing only disaster recovery. Note that it is the fact that a filesystem may be reloaded from backup tapes that forces the secondary storage copies of all migrated files to be retained for an aging period. This ensures that if the most recent backup is reloaded onto the disk, the secondary storage copies of recently deleted files will continue to exist for a brief additional time, so that they can be reloaded by the user if needed.

The migration system does not keep a consistent set of file names on the secondary storage media, because all activity is identified by the inode number and file handle, and because one inode may have many different path names that lead to it. While the migration system does record a complete copy of the information in regular file inodes onto the secondary storage, it does not record any path name information, because directories and special files can not be migrated. As a result, the migration system is not suitable for use as a filesystem backup mechanism. If a filesystem was entirely destroyed and no dump tapes were usable, enough information is stored in the migration system so that all migrated files could be recovered, but all files would appear in their owners home directory.

Having the file migration capability does not relieve the need for a user tape handling facility, nor does it permit users with large storage requirements to ignore their responsibility for monitoring their storage usage and managing their collection of private tapes. The file migration system merely changes the point at which the filesystem appears to be full. The limiting factor will shift from the number of disk blocks available on a filesystem to the number of inodes available on a filesystem. Most UNIX filesystems have a fixed number of inodes allocated per filesystem at the time that the filesystem is initially created (with **mkfs**). Thus, the additional inode requirements must be taken into account when filesystem are created. Many more inodes will be needed on filesystems that will store migrated files; 200,000 inodes is a good minimum for larger filesystems (0.5 Gbytes to 1 Gbytes).

Even those few UNIX systems that can dynamically allocate additional inodes when needed are still limited by the fact that the size of the disk provides an upper bound on the number of files on the filesystem. This seemingly contrived limiting case becomes more of a serious issue when online storage for at least one copy of the largest user file needs to be held in reserve from the disk full of inodes. Failure to observe this relationship could prevent users from retrieving their largest files, and in general would probably be preceded by the migration system being forced into severe "thrashing" of files between primary and secondary storage.

Therefore, when implementing the file migration system on a particular machine, it is strongly recommended that the administration adopt a two-part policy on file storage, and that the user community be advised of this policy before the file migration system is activated. The first part of the policy is to note that when online space is required, user files may be transparently moved to secondary storage, following the detailed rules of the migration policy elected by the site. The second part of the policy is to note that files may not be maintained in the migration system, even on secondary storage, for more than a period of 1.5 years (or a similar time limit), and that users who require longer term storage of their data must take advantage of the user tape handling facility that has been provided. An automatic tool will be provided that will, with suitable advance warning to the users via E-mail, enforce this policy.

## 14. CURRENT STATUS

As of this writing, the kernel support for the migration system is complete, and has been well tested. A demonstration package which includes **migout**, **migin**, and the migration daemon has been assembled that allows files to be migrated and unmigrated. Implementation of the **migarch** utility and the set of methods for tape-style devices is well underway. Overall, most of the pieces now exist, and the final work of assembling the high-level tools is progressing well.

It is anticipated that this software will be in full production status in BRL in the early Fall of 1988, and will be made generally available as Public Domain Distribution Unlimited software by Fall/Winter of 1988.

## 15. FUTURE WORK

The task of moving a user from one filesystem to another filesystem, to more evenly balance storage requirements is a task that system administrators will occasionally have to perform. Typically, this is accomplished on Berkeley systems using back-to-back **tar** programs, eg,

```
cd fromdir; tar cf - . | (cd todir; tar xf -)
```

and on System V machines, this is done using a combination of **find** and **cpio**, eg,

```
find . -depth -print | cpio -pdlm
```

If a user with a significant number of migrated files was to be relocated to a new filesystem using this technique, the correct effect would be achieved, but the migration system would engage in a significant amount of activity. All the files on the source filesystem would have to be migrated in to the original disk, that would probably cause the space management function to have to out-migrate other files to make room. Then, all those files would be copied to the destination filesystem, that would probably also cause the space management function to have to out-migrate files there, too. The result of this is that whole collection of old files will have been brought back onto disk on a new filesystem, where they will consume online storage until they have aged sufficiently to qualify for out-migration. Adequate kernel support exists to permit the implementation of a **cpio -p** substitute that would relocate migrated inodes without forcing a reload operation, but as of this writing, this has not yet been done.

For the majority of UNIX systems, the most popular secondary storage medium for file migration is likely to be operator mounted reels of magnetic tape. This makes the interface to the operator an important part of the file migration software, so that it is easy for the operator to determine what tape is to be mounted, and on which drive. In order for the migration system to be effective, this must work well, or the extra delay in mounting tapes will reflect poorly on the migration system. However, the design and implementation of a good operator interface mechanism for UNIX is properly the subject of another project. By necessity, the initial version of the file migration software will offer a very simple interaction with the operator, except on systems like the Cray where an operator interface package already exists. Then, as time progresses, a separate effort to implement a good, portable operator interface package will be initiated, with the goal being to release another piece of software into the public domain. For the present, this has not been done.

Many additional migration methods will be added as the project progresses, with network tapes, Masstore systems robotic cartridge tape units, and the 8mm Exabyte tape cartridges being prime candidates.

The current design accomplishes its goal of no filesystem inode changes; however, this prevents certain worthwhile features from being provided. For example, a desirable feature is the ability to reclaim space used by files that have been reloaded for read-only access, by taking advantage of the copy on secondary storage. Another is to be able to supply the first portions of a file to a reading process while the balance of the file is being reloaded. These features require that the filehandle and quantity of reloaded data be added as new fields in the filesystem inodes. Future work will incorporate these filesystem changes and provide enhanced functionality.

In order to minimize the impact on the kernel, and various ancillary programs such as **mount**, the current implementation uses a single low space threshold, expressed in blocks, for all mounted file systems, regardless of file system size. A much better strategy would be to read the low space threshold from a file such as /etc/fstab, and provide it to the kernel as part of the **mount**(2) system call. This would allow the low space threshold to be set independently for each filesystem.

In a network filesystem environment, such as provided by Sun NFS, the migration software will reside entirely on the file server machine, so that file reloading will be entirely transparent to the client machines. It is presently unknown whether there will be any interactions between the potentially lengthy time required to open a migrated file, and protocol timeouts in the client machines. It may also be necessary to increase the number of **nfsd** daemons to account for some of them being blocked on migrated file opens.

Presently, there is no way to add the functionality of the new system calls **mig_stat()** and **mig_lstat()** to the current NFS protocol. One implication of this, combined with the fact that the file server handles the migrated files, is that there is no way for a client machine to determine whether a file is migrated or not. Finally, it is not clear how to propagate signals on the client to the server. If the an open of an NFS file is blocked on the server due to a migration reload operation, the signal needs to travel over the network. All these issues will be investigated in detail after the software is operating well for machines with locally attached disk drives.

## 16. CONCLUSIONS

With a very limited set of modifications to the UNIX kernel, it is possible to provide a fully transparent file migration capability, preserving the complete semantics of the UNIX filesystem. Built around these kernel capabilities are a highly modular set of utility programs that select files for migration, and implement the details of moving files between different devices. One of the best aspects of the design is the strong separation between *policy* and *mechanism*, so that the special needs of individual sites can be satisfied with a single software mechanism.

This software promises to satisfy a need that large scale users of UNIX systems have long felt, yet provides the capability in such an elegant and transparent manner that even the most discriminating UNIX person should not bristle. Much.

### Acknowledgements

The authors would like to thank Chris Johnson for his careful and thorough review of the kernel code, Rick Matthews for his stimulating interchange of ideas, and Bob Reschly and Phil Dykstra for their advice, proofreading, and gratuitous humor as we designed and built this software.

The following strings that have been included in this paper are known to enjoy protection as trademarks; the trademark ownership is acknowledged in the table below.

| Trademark | Trademark Owner |
|---|---|
| Cray | Cray Research, Inc |
| Ethernet | Xerox Corporation |
| FX/8 | Alliant Computer Systems Corporation |
| IBM 370 | International Business Machines Corporation |
| Macintosh | Apple Computer, Inc |
| NFS | Sun Microsystems, Inc |
| PowerNode | Gould, Inc |
| ProNet | Proteon, Inc |
| Sun Workstation | Sun Microsystems, Inc |
| UNIX | AT&T Bell Laboratories |
| UNICOS | Cray Research, Inc |
| VAX | Digital Equipment Corporation |