Persistent Shared Heap File System (PSHFS)

Abstract

Fast and efficient access to data storage is an important concern in the modern computer industry. One of the problems is that the only way to share large amounts of data across applications is to serialize it and store it on disks. As an alternative we describe the design, implementation, and evaluation of a new file system model that uses shared memory as a storage partition, eliminating the need for expensive I/O system calls such as read and write, simplifying programming when sharing and storing of large complex data structures is required, and avoiding the need to copy data through multiple buffers. The main idea behind the new file system implementation is to expose a malloc-like programming API, which permits programmers to create and operate directly on shared complex objects residing in the file system. At the same time full backward compatibility is maintained, preserving standard file system behavior with regard to regular files. While our current implementation is not persistent between system reboots, we note that it foreshadows interesting developments that will be possible with future non-volatile memory technologies.

1. Introduction

The Unix locate utility is used to locate files in the system, typically libraries and other such system files. It works in two phases. The first is creating a database of all the system's files using the updatedb utility. The second is looking for files in this database with the locate utility. Using a pre-compiled database leads to much faster search times relative to the find utility, which actually scans the file system when invoked. locate also has flags to verify that the files listed in the database indeed still exist, as the database may be out of date, but using them naturally takes more time.

The database of file names is naturally kept in a file. To conserve space, it is compressed using an incremental encoding scheme, where filenames are sorted and each one is represented by its difference from the previous one. This means that file search must read the database sequentially, which leads to an efficient sequential access pattern to the database file.

An alternative data structure that may be used to store and search for file names is a hash table. Hash tables are more efficient because the expected search time is constant, rather then depending on the size of the dataset. However, using hash tables is not applicable in the context of locate, because they would have to be created by one application (updatedb) and used by another (locate). This would imply that updatedb has to serialize the hash table and store it in a file, and locate then needs to reconstruct it, thereby losing all the performance advantages.

The above situation is not unique to locate – it applies to any applications that create and use complex data structures. One example is applications that use graphs, such as entity-relationship models, which can be used to describe any ontology for a certain universe of discourse. Another is computational geometry and image processing, where image data may be stored in a quad-tree that is expanded according to need [Munroe 2007]. It also applies to the checkpointing of complete applications, where all an application's data structures need to be efficiently stored and restored.

Another major problem with access to stored data in modern operating systems is that it requires multiple data copies. These data copies, especially ones from a disk drive, are very expensive. For example, the read command data flow in an application running on a modern operating system first involves copying the data from the disk device to a kernel buffer in main memory, typically using DMA, and then copying the data to the buffer specified by the user. This second copy is done by the processor, wasting CPU cycles and possibly disrupting cache state.

To solve the above problems we introduce PSHFS, a kernel module add-on based on the traditional shared memory file system (tmpfs), which allows applications to take better advantage of the system's memory management facilities. A new approach is introduced, providing a framework for improving efficiency of the I/O and memory management mechanisms. The idea is that memory objects created using a new malloc-like API can be turned into files as is, without being serialized and copied through multiple buffers. They can then be attached to other applications, which can immediately use them, saving the need to reconstruct the object from the serialized representation.

At this point it might seem that the PSHFS functionality is the same as that of shared memory segments. However, shared memory is different in several respects. First, it uses a different namespace, thus losing the uniformity that comes from sticking to the file system namespace. In addition, the implementation is based on mapped files, that suffer the overheads of buffer copies similar to using the read and write system calls. But most importantly, when using shared memory one still needs to serialize complex data structures, because pointers would become invalid if the structure is mapped by another process at a different location. PSHFS avoids this problem by using named pointers, i.e. pointers that use the file system namespace. Thus PSHFS in effect integrates the shared memory functionality with the conventional file system to obtain efficiency and ease of use.

Our implementation of PSHFS is based on the Debian operating system. Debian OS is based on the standard Linux kernel, allowing this work portability between other flavors of Linux, and is used widely for OS research. Our implementation provides persistent storage between unrelated applications. However, it is not persistent across system reboots. This is a design choice, which may well be rectified in future versions. Alternatively, one may view our prototype as foreshadowing developments that will be possible with future non-volatile memory products such as phase-change memory [PCM] that are not as restrictive as current flash technology. Such non-volatile memory will also offer full persistence.

The rest of this paper is organized as follows. The next section presents some technological background. Section 3 presents the design and implementation of PSHFS, and Section 4 its performance. Finally, Section 5 reviews related work, and Section 6 concludes the paper.

2. Background

PSHFS is based on tmpfs and VFS, and is obviously related to malloc and shared memory. We therefore provide a brief review of these topics before going into the details of PSHFS.

2.1 Memory File Systems

Virtual memory based file system such as tmpfs [Snyder, McKusick] are similar in concept to RAMdisks, in that they use RAM instead of disk to obtain better performance. The difference is that a traditional RAMdisk is a block device, whereas tmpfs is a complete filesystem. In addition, RAMdisks typically use a pre-allocated dedicated block of memory, while tmpfs sits on top of virtual memory (VM) and may therefore use both RAM and swap. The VM subsystem allocates RAM and swap to various parts of the system, and takes care of managing these resources behind-the-scenes, often transparently moving RAM pages to swap and vice-versa. The tmpfs filesystem requests pages from the VM subsystem to store files and doesn't know whether these pages are on swap or in RAM.

The size of the tmpfs file system can be dynamically increased – the tmpfs driver will allocate more VM and will dynamically increase the filesystem capacity as needed. And, as files are removed from tmpfs, the driver will dynamically shrink the size of the filesystem and free VM resources, and by doing so return VM into circulation so that it can be used by other parts of the system as needed. tmpfs data is not preserved between reboots, because virtual memory is volatile in nature.

The PSHFS implementation uses tmpfs as a base, adding a malloc-like API to the existing code in order to simplify programming, and preventing multiple buffer copies by mapping the actual file system pages into process memory.

2.2 VFS

Due to filesystems, applications no longer have to deal directly with the physical storage medium. But the Linux operating system supports multiple different filesystems. To enable the upper levels of the kernel to deal equally with all of them, Linux defines an abstract layer, known as the Virtual File System, or VFS.

Each lower level file-system must present an interface which conforms to VFS. This interface is structured around a number of generic object types, and a number of methods which can be called on these objects. The basic objects known to the VFS layer are files, file-systems, inodes, and names for inodes.

Files are streams of bytes stored as a single unit, which can be read from or written to.

Inodes represent basic objects within a file-system, e.g. a regular file, a directory, or a symbolic link. VFS itself does not make a strong distinction between different types of objects, but leaves this to the actual file-system implementation.

File Systems are a collection of inodes with one distinguished inode known as the root. Other inodes are accessed by starting at the root and looking up a file name to get to another inode. Each file-system resides on a unique device, but some (such as nfs and proc) don't need a real physical device.

Names are used to access inodes. Names are given relative to a directory, leading to a hierarchical namespace.

dcache is a cache for currently active and recently used names, structured in memory as a tree. Each node in the tree corresponds to an inode with a given name in a given directory, so an inode can be associated with more than one node in the tree.

dentry is an entry in the dcache, and acts as an intermediary between open files and inodes.

2.3 Malloc

The malloc command is the workhorse of dynamic memory allocation. Such memory is allocated from the heap, and is not sharable with other processes.

An important aspect of dynamic memory allocation is the management of heap space, and trying to prevent fragmentation. In particular, allocations of less than a full page (e.g. for new objects in object-oriented programming) must be supported efficiently. The PSHFS prototype does not support efficient small allocations, but this is a technical issue that can be solved with sub-paging techniques [Itzkovitz 1999] more than an inherent barrier.

2.4 Shared memory

Linux processes typically do not share memory – in fact, one of the roles of the system is to isolate processes (and their address spaces) from each other. But it is also possible to share a memory segment. This is done by mapping the same segment into the address spaces of the sharing processes.

Shared memory segments are created by shmget, which specifies their name and access permissions. They are then attached to the sharing processes using shmatt. When not needed, they should be removed from the system using shmctl; if not removed this is considered a memory leak. PSHFS is different in regarding such behavior as a feature, and allowing data structures to be retained for long periods. Moreover, the shared memory objects can be given names that make them appear as regular files.

3. PSHFS Design and Implementation

VFS is an indirection layer used to handle system calls acting on files located on traditional file systems. This indirection mechanism is used by the Linux operating system to allow use of several filesystem types. When a file oriented system call is issued, the kernel calls a function contained in the VFS. This function handles the structure independent manipulations and redirects the call to a function contained in the physical filesystem, which is responsible for handling the structure dependent operations.

Here we introduce PSHFS (Persistent Shared Heap File System), a novel virtual memory filesystem for Linux, that makes better use of existing hardware memory management features to reduce overhead and improve performance. PSHFS is implemented within the framework of the Linux kernel, and maintains full backward compatibility to the traditional virtual memory filesystem (tmpfs), which was used as a base for the implementation. PSHFS supports UNIX file semantics and provides file system space, based on shared memory.

3.1 Basic Ideas

The main idea behind PSHFS's implementation is to expose a malloc-like programming API, which permits programmers to create and operate on persistent, shared objects (a.k.a. files) on the file system, without using expensive system calls, such as read and write. On the other hand, full backward compatibility is maintained, preserving standard file system behavior in regards with regular files. Such an API simplifies programming by creating a single level of abstraction (files = objects), and maximizes data efficiency by preventing multiple buffer copies.

PSHFS	**Tmpfs**
Dereference object name to obtain pointer	Open file to obtain file descriptor
Map process memory to object memory	*Copy data from physical storage to kernel buffer*
	Copy data from kernel buffer to process buffer
	Repeat until all data has been copied
	Deserialize data read to reconstruct object
Access object data using pointer	Access object data using pointer

Table 3-1 read() comparison.

Table 3-1 demonstrates this by comparing the overhead of using PSHFS to that of a conventional read system call (time taken by steps marked in *italic* is a function of file/buffer size, time taken by other steps is constant).

As indicated in this example, the PSHFS implementation goals are achieved by basically mapping files space (objects) into process address space. A new definition of the term "file" is provided, where serialized data access is no longer used. Instead data is stored in its original ("object") format, and data original structures are preserved.

Since data has to be persistent after process termination, anonymous mmap is no longer used. To preserve the naming and hierarchical structure of the file system, we introduce name pointers (unique names) and file names. Persistent name pointers are automatically generated for all objects, and a new file system module protects from race-conditions during object name selection. In addition, objects can be assigned user-chosen file names. Objects can then be referenced by either these object names or file names, providing functionality similar to the original file system's namespace.

The common approach to physical memory based file systems (RAMdisks) is to reserve a chunk of physical memory for use of the file system. RAMdisks use memory inefficiently, since data is duplicated on file system memory and in kernel memory. In contrast, PSHFS uses the memory more efficiently. No memory-to-memory copy is required, eliminating the need for most resources consuming system calls such as reads and writes.

To summarize, PSHFS deals with the following drawbacks of traditional file systems in the following ways:

- Performance degradation due to multiple buffer copies, especially copies between kernel space and user space buffers: PSHFS provides direct access to files, so no intermediate buffers are required.
- Unnecessary overhead due to multiple system calls, such as read: PSHFS eliminates the need for such system calls
- Data duplication – the same data is stored in memory and on disk, which, in the case of a virtual memory file system, leads to unnecessary duplication: again, PSHFS avoids this by providing direct access to the files themselves.
- Serialization is required for data structures to be stored on disk – objects have to be serialized when written to the file system and de-serialized when read from the file system: PSHFS allows the persistent data to be stored in its original object format, with links based on named pointers, so no serialization is required.

3.2 Architecture

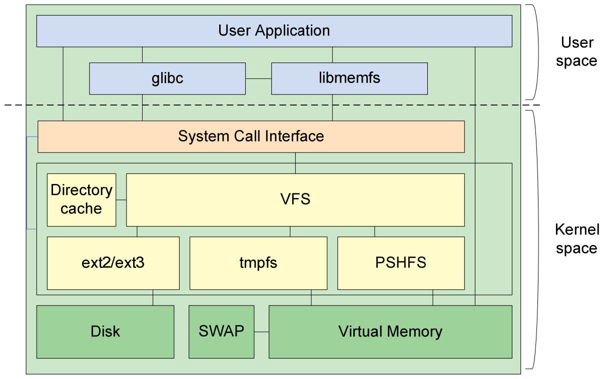

Figure 3-1: System Architecture

PSHFS is comprised of two basic components , as illustrated in Figure 3-1 (the implementation is called memfs, which stands for MEMory File System):

- Loadable kernel module, or "memfs kernel module" – an extension of the tmpfs kernel module, with a number of modifications, to allow PSHFS operation.
- An API library, or "libmemfs library" – an abstraction layer and interface to PSHFS services.

The precise functions embedded in each component will be listed later, after we describe the API.

Upon PSHFS module load, the new file system type is registered with the kernel, and mounted, if required, according to the regular VFS procedure.

PSHFS is compliant with the standard virtual filesystem (VFS) abstraction layer. The VFS access interface is used to acquire file descriptors of PSHFS files, in order to perform actual shared memory mapping into the process space. Once mapping is performed, the file descriptor is no longer used and is released.

PSHFS can be used also as a regular shared memory file system (tmpfs). In this case its operation is identical to that of tmpfs.

The basic structures used within PSHFS are:

- **mfobj**, which represents an object (file). An object is mapped into a process address space, and accessed via a standard pointer (ptr). In addition, a named-pointer (nptr) represents the object's automatically generated name, which is used as a permanent identifier of the object in the objects store (file system). Objects can also be attached to an unlimited number of user-generated names (represented by symbolic links). The role of mfobjs is illustrated in Figure 3-2.

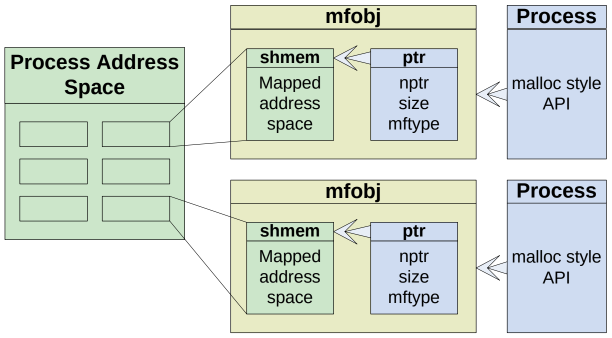

Figure 3-2: Basic structures

- **mftype**, which is used to describe the internal structure of an object, by providing a list of offsets to named pointers (nptr's) within it. This is used only when a process calls mfclone(), which creates a deep copy of an object, so pointers to referenced objects need to be found.

3.3 Basic API

The following is a simple example of using the PSHFS malloc-like API. All these functions are explained below.

```
mfobj *mfobj;
int error = mfopen("/mem");          //Attach  filesystem
mfobj = mfalloc(1000);               //Allocate object
nptr *namedptr = mfobj->nptr;        //Get  object's  name
char *ptr = mfdereference(mfobj);    //Get pointer
memset(ptr, '0', 1000);              //Initialize
mfaddname(mfobj, "FILE1");           //Give it a name
mfcloseobj(mfobj);                   //Detach object
mfobj = mfgetobjbyname("FILE1");//Get by name
mffreeobj(mfobj);                    //Delete
error = mfclose();                   //Detach file system
```

mfopen()
mfopen("path") opens the file system to access the name space. This can be done on any directory within PSHFS, but only a single namespace (directory) can be attached to a process at a time. The following operations are performed when a namespace is accesses with mfopen():

- A balanced red-black tree is created, and used to store objects (attached to the current process) information. The red-black tree mechanism is used to provide quick O(lg(n)) mapping between nptr's (named pointers) and ptr's (regular malloc-like pointers), for the files attached to the current process. A possible alternative attached objects data structure is a hashtable.
- The file system is verified to be PSHFS compliant.

mfalloc()
mfobj = mfalloc(int size) allocates an object, with size specified, utilizing the PSHFS malloc-style API. This operation creates an mfobj struct, and generates a unique name (nptr) for the object. nptr also represents a physical file in the underlying tmpfs-like file system.

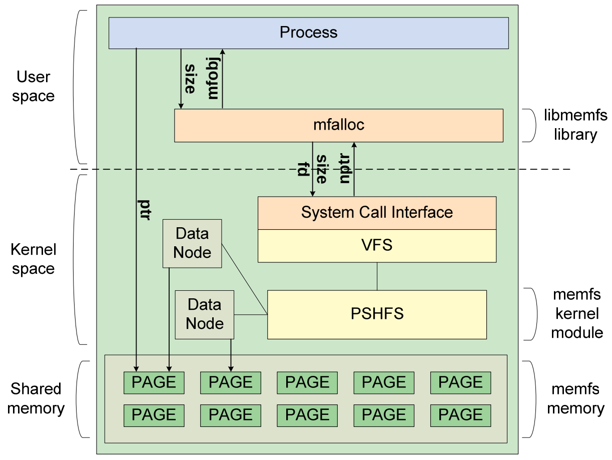

Figure 3-3: mfalloc()

Object initialization is performed in the following steps:

- Verification that the designated file system is PSHFS compliant
- Create a file descriptor (and empty file with requested size). An nptr is generated automatically for the file created.
- Map shared memory segment with requested size into process memory space, and acquire a pointer (ptr) to the mapped pages.
- Close the file descriptor
- Register object in objects red-black balanced tree.

mfdereference()
**ptr = mfdereference(mfobj)* provides a pointer, which can be used to access an object's memory directly. This can be done only on objects already attached to the current process.

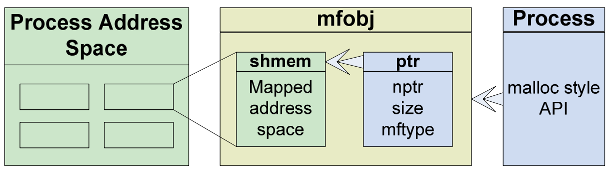

Figure 3-4: mfdereference()

Once the pointer (ptr) to the object's storage is acquired, regular pointer API operations can be performed, so *memset(ptr, '0', 1000)* will fill the first 1000 bytes of the memory area pointed to by ptr with constant byte 0. To appreciate the power of direct access, consider the following tmpfs code:

```
write(fd,"00000",5);
lseek(fd,0,SEEK_SET);
read(fd,buffer,5);
fprintf(stdout," Current Value:  %s\n",buffer);
lseek(fd,0,SEEK_SET);
write(fd,"11111",5);
lseek(fd,0,SEEK_SET);
read(fd,buffer,5);
fprintf(stdout," Current Value:  %s\n",buffer);
```

using PSHFS, this turns into

```
memset(ptr,'0',5);
fprintf(stdout,"Current Value:  %s\n",ptr);
memset(ptr,'1',5);
fprintf(stdout,"Current Value:  %s\n",ptr);
```

seeking to a location other than 0 would translate to adding an offset to ptr.

mfaddname() & mfgetobjbyname()
mfaddname(mfobj, "filename") allows addition of human-provided names to the object. Initially, an object is automatically given a named pointer (nptr) file name, but in order to provide a common interface to file naming, and stay in line with file system acceptable naming policies, it is possible to add any number of names to a given object. Additional file names are

represented by symbolic links on the file system, which are automatically removed if the object is freed or deleted. To enable this, the symbolic links are listed in the object's inode (see below).

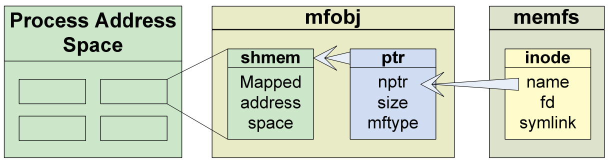

Figure 3-5: mfaddname()

mfobj = mfgetobjbyname("filename") opens an object pointed to by the given filename and attaches it to the current process memory address space (just like mfalloc, except that the object does not need to be created).

mfgetobj() & mfnptr()
mfobj = mfgetobj(nptr) obtains the mfobj pointer to by a given nptr.
nptr = mfnptr(mfobj) performs the opposite mapping, extracting the nptr from an mfobj.

mfcloseobj()
mfcloseobj(mfobj) detaches the object from the current process, but leaves it in the storage space. The object can be reopened and used at any time.
An object is unmapped in the following steps:

- Object shared memory block is unmapped from current process address space
- Object is removed from process's red-black balanced objects tree.

mffreeobj()
mffreeobj(mfobj) detaches the object from the current process, removing it from the storage, and freeing the shared memory taken by the object. In addition, all symbolic links pointing to the object are removed. The memfs inode structure has an additional field, "*symlinks*", containing a list of pointers to nptr's of the symbolic links (pointers to inodes of the symbolic link files) to the object being removed. Symbolic links are removed according to this "*symlinks*" list.

mfclose()
mfclose() detaches the PSHFS name space, opened with mfopen. The following operations are performed when a namespace is detached with mfclose():

- Red-black tree, containing objects mapping, is freed.
- Namespace is destroyed

3.4 Advanced Functionality

mfcopy()
copy = mfcopy(mfobj) creates a copy of a SINGLE object. Objects referenced by *mfobj* are not copied. This operation creates a new mfobj struct, and generates a unique name (nptr) for the object, in a way similar to the *mfalloc()* function. nptr also represents the physical file on the underlying tmpfs file system.
Single object copy is performed in the following steps:

- Perform mfalloc() for a new (destination) object with size similar to that of the source
- Copy using memcpy all data from the source to the new object

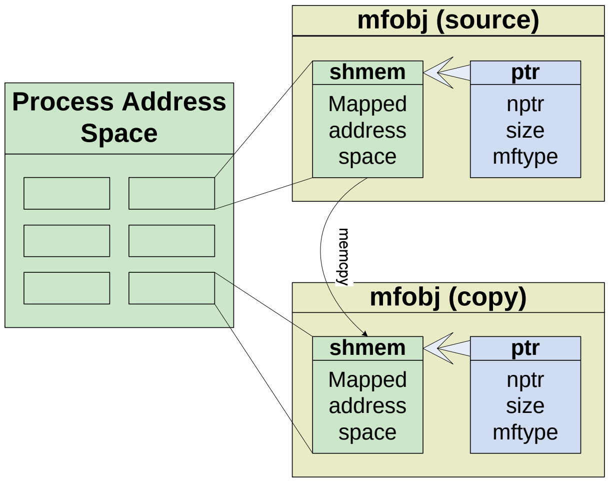

Figure 3-6: mfcopy()

mfalloctype() & mftypeatt()
We use a "types" method in order to deal with complex object operations, such as deep-copy. "type" is represented by an array attached to a complex object, and containing offsets of all named pointers within the given object. A single "type" can be attached to multiple objects with the same structure. For example, given the following complex object, representing a tree node:

```
typedef struct {
        nptr *left;             // Pointer to left child
        nptr *right;            // Pointer to right child
        int val;                // Node value
} tree_node;
```

will require the following "type" structure, in order to be deep-copied (cloned):

```
size_t type[] = {0,4};      // Array of offsets to pointers
mftype *mftype;             // Types pointer
mftype = mfalloctype(type, 2);//Alloc and init
mftypeatt(mfobj, mftype);   // Attach mftype to object
```

In this code, type is the array of offsets within the struct: 0 = left, 4 = right. Rather than hardcoding, it is also possible to use the offsetof macro in order to determine the offsets. mfalloctype() takes such an array of offsets and its size and converts them to an mftype, which is finally attached to the object itself.

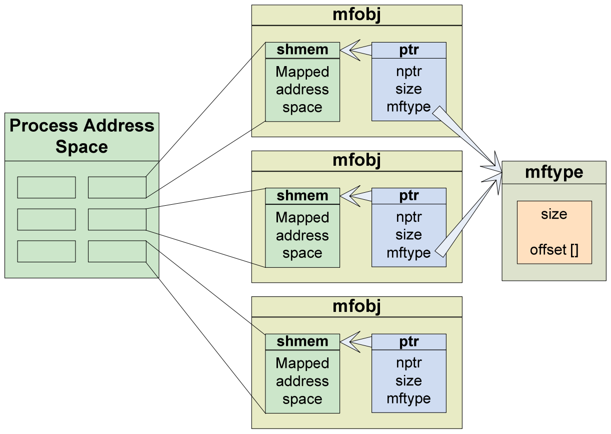

Figure 3-7: attaching an mftype

mfclone()

clone = mfclone(mfobj) creates a copy of a whole object tree (deep-copy). All objects referenced by *mfobj* are copied. All referenced objects must be identified in the mftype attached to each of the complex objects in the objects tree to be copied.

Cloning of an object tree (object and objects referenced from it) is performed in the following steps:

- Selection of the sub-graph of objects to be cloned graph from the graph of all objects. An acyclic, connected, directed graph of the object is built, based on offset information located in mftype structures. Mftype structures are recursively analyzed in order to build the graph.
- Correction of cyclic links to prevent dead end loops.
- Each object in the selected sub-graph is copied and new nptr's are allocated.
- Correction of new nptr's to reflect the copied objects.

As an example of all this, here is code that handles a tree with a root and two sons (see Figure 3-8):

```
// Create tree
mfobj *root = mfalloc(sizeof(tree_node));
tree_node *root_ptr = mfdereference(root);
root_ptr->left = mfnptr(mfalloc(sizeof(tree_node)));
root_ptr->right = mfnptr(mfalloc(sizeof(tree_node)));

mfaddname(root, "THIS_IS_A_TREE");

// Assign values
tree_node *left_ptr =
        mfdereference(mfgetobj(root_ptr->left));
tree_node *right_ptr =
        mfdereference(mfgetobj(root_ptr->right));
root_ptr->val = 1;
left_ptr->val = 2;
right_ptr->val = 3;

// Detach objects
mfcloseobj(mfgetobj(root_ptr->left));
mfcloseobj(mfgetobj(root_ptr->right));
mfcloseobj(root);

// Re-attach (just for the example)
root = mfgetobjbyname("THIS_IS_A_TREE");
root_ptr = mfdereference(root);

// Make a copy of root node only
mfobj *copy = mfcopy(root);
mfaddname(copy, "COPY_TREE");

// Clone the whole tree
size_t type[] = {0,4};
mftype *mftype;
mftype = mfalloctype(type, 2);
mftypeatt(root, mftype);
mftypeatt(mfgetobj(root_ptr->left), mftype);
mftypeatt(mfgetobj(root_ptr->right), mftype);
mfobj *clone = mfclone(root);
mfaddname(clone, "CLONE_TREE");
tree_node *clone_ptr = mfdereference(clone);
```

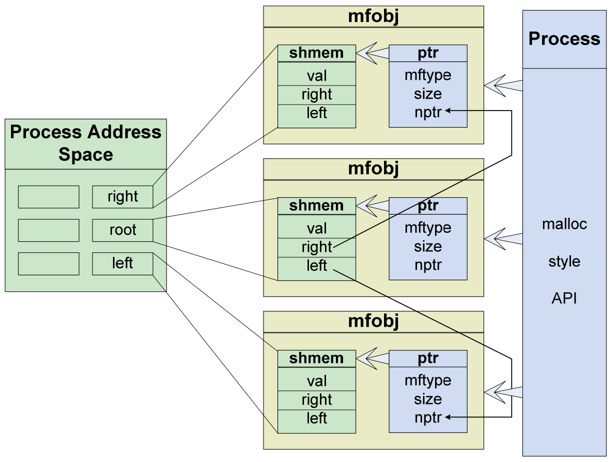

Figure 3-8: Tree Sample

3.5 Sharing Data

Naturally more than one process may attach the same objects to its memory space, leading to data sharing. This requires some handling of security and synchronization.

Standard file system permissions hierarchy and rules apply. For regular files, no file descriptor can be acquired without appropriate permissions set on the file. For PSHFS objects, shared memory will not be mapped into process address space, unless right permission are set on the object. In the current implementation there is no read-only mapping provided for read-only objects, and memory pages are always mapped in a RW mode. In order to resolve this permissions issue, it is possible to add a parameter specifying the object's permissions to the mfalloc library call, or to allow manual setting of a default file mask, such as with the umask() function.

PSHFS provides full backward locking compatibility for regular files (such as fcntl() and flock() mechanisms). For PSHFS files, a standard shared memory semaphore-based mechanism is recommended:

- semget function can be used to grab a semaphore
- semop function to test-n-set a semaphore
- semun function to destroy a semaphore
- semctl function provides a variety of semaphore control operations
- ftok function returns a key based on path and id that is usable in subsequent calls to other semaphore functions

3.6 Implementation

As noted above, the implementation is based on two components: a loadable kernel module and a user library.

The PSHFS kernel module is an extension of the standard Linux tmpfs module, and it is used to introduce the actual PSHFS (memfs) file system into the Linux kernel. A number of modifications to standard tmpfs were made to provide support for desired features:

- **Directories namespace** – each directory in PSHFS provides a separate namespace (when libmemfs is concerned), therefore to avoid race-conditions on a given PSHFS filesystem, unique names generation for files is provided on the kernel module level, and not from libmemfs. The kernel module provides a single system call, ioctl IOMF_NEW, which generates a new name for the new object (file), and registers the object on the filesystem.
- **File names** – in addition to a regular, serialized nptr (named pointer) file name, it's possible to add a number of strings or "regular" names to the file (implemented using symbolic links). When an object (file) is removed, its "regular" names are removed automatically as described above.
- The file system utilizes the **directory cache** (dcache) to store the directory structure, increasing dentry counter on its creation, which forces the dentry to stay in cache. The dentry references an inode, allocated in slab_cache (kmem_cache_t).
- **memfs_inode**, besides standard vfs_inode data, contains two additional fields:
  - o Symlinks which is used for automatic removal of real names (symlinks) pointing to this inode's PSHFS file.
  - o Dentry, pointing to dentry which owns inode of given symbolic link.
- **Hard links** (to nptr) are not allowed in PSHFS.
- **Introducing "t" bit** – files generated via ioctl(IOMF_NEW) are marked with special bit "t" (stored in the inode), in order to provide differentiation between "objects" and regular "files". The "t" bit is validated during API functions execution on files.
- **Filesystem type verification** – the kernel provides a system call ioctl(IOMF_NEGUINT), which can be used to verify that a given filesystem is of type memfs (PSHFS).

The PSHFS library (libmemfs) provides an abstraction layer and interface to PSHFS services, providing as well some additional functionality, such as:

- Malloc-style API (described in detail in Section 3.3)
- Types subsystem
- Recursive copy of objects graphs
- Listing of objects attached to the process address space
- Balanced red-black tree storing objects information per process

The malloc-style API provides standard wrappers for malloc system calls. One of the most important features of the new API implementation is the fact that file descriptors are released when object operation is complete, which on one hand increases the number of system calls, but on the other hand allows a large number of objects to be connected to a process without overrunning the limit of 1024 file descriptors per process.

The "types" subsystem provides a list of references to other objects inside a specific object. This list of offsets is used when a copy of the object tree is required. Copy of object tree is implemented in user-space, therefore types are not attached to actual objects on the file system. The reason for performing the object-tree copies in user-space is based on the fact that performance degradation can be caused by copying large objects in the kernel space, while implementation of copy-on-write doesn't exist in VFS scope.

In order to insure object structure is kept (according to mftype), and to disallow unwanted pointers modifications from user-space (using for example standard OS utilities), it is not allowed to reduce the size of an existing object. Such implementation is dictated by the fact that information regarding pointers location in a memory block is not passed to the kernel, but exists only in the name space of libmemfs.

The current implementation has several drawbacks that should be noted:

- PSHFS is a plug-in to the existing tmpfs file system. Such dependence creates additional overhead – for example, a file descriptor has to be obtained in order to create the actual memory mapping.
- PSHFS size is limited by shared memory size, in contrast to real file systems, where limitation is actual size of the disk.
- PSHFS file size (and total size of the file system) is limited by 4GB in total (32 bit).

- Fixed PAGE_SIZE creates overhead on small files.
- Persistence across power cycles is not yet supported.

4. Performance Results

We evaluated the performance of PSHFS first using microbenchmarks, and then by implementing a version of locate that uses PSHFS to store its database.

The test environment was in Intel-based PC with a Core2 Duo processor running at 2.20GHz and equipped with 1GB of RAM and a 160 GB disk. The operating system was Debian release 4.0 with a 2.6.18.6 standard Linux kernel.

4.1 Micro Benchmarks

The micro benchmarks are designed to measure the performance of individual operations. We measured the performance of sequentially reading data, sequentially writing it, random reading interspersed by seeking operations, and copying. This was repeated for different file sizes from 1KB to 2MB with the size doubling at each step, and using buffer sizes of 1, 64, 512, and 1K bytes (in the random access measurements, only a buffer size of 512 was used).

The results (Figure 4-1 to Figure 4-4) show that PSHFS is largely insensitive to the buffer size used: the results for read and copy are identical for all buffer sizes, and for write small buffers suffer a relatively small degradation. The standard file system suffers a much larger degradation in performance for small buffers, due to the constant overhead involved in the system calls. On the other hand PSHFS suffers from a constant overhead for small file sizes.

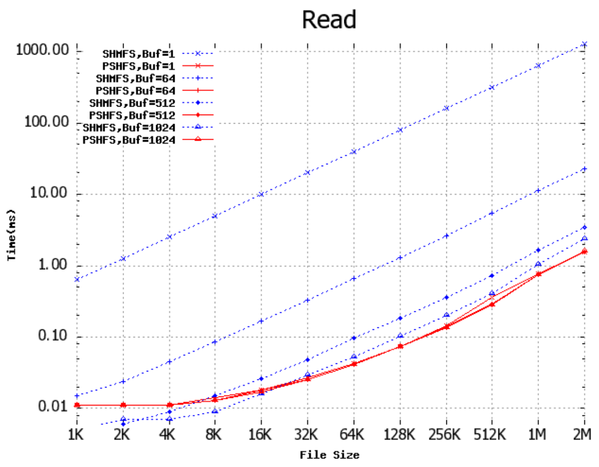

Figure 4-1: Read

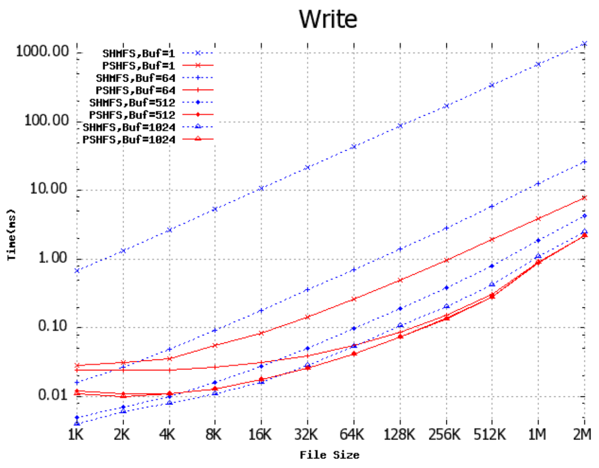

Figure 4-2: Write

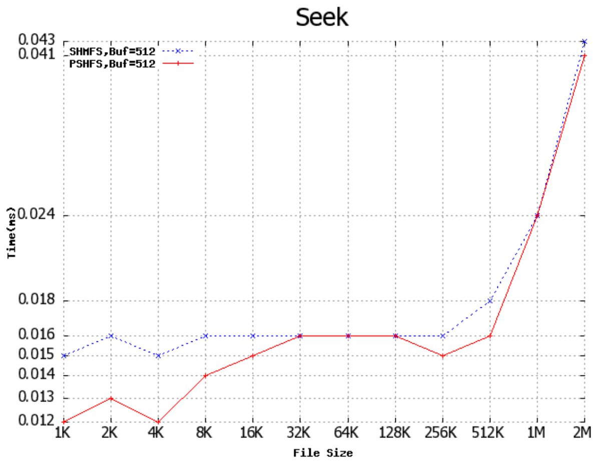

Figure 4-3: Seek

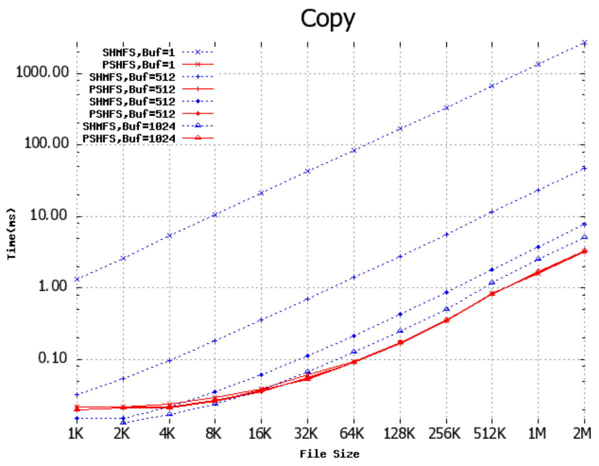

Figure 4-4: Copy

4.2 Locate Utility

The Unix locate utility mentioned in the introduction was selected to demonstrate the massive performance increase when using PSHFS to store complex objects. A simplified version of the slocate flavour of locate was used as a base for the benchmark implementation. slocate (Secure locate) [slocate] provides a secure way to index and quickly search for files on the system. It uses incremental encoding (similar to GNU locate) to compress its database to make searching faster, but also checks file permissions and ownership so that users will not see files they do not have access to.

As part of the benchmark preparation, several mixed directory/file structures were created on a regular physical partition. Directory structures containing 100, 200, 300, 500, 1,000, 2,000 and 4,000 entries were created.

Rather than timing the whole utility, we instrumented both the original slocate code and the PSHFS version, to allow performance analysis of relevant code parts only. The code parts that were measured are the creation of the database and the search function.

The PSHFS flavor of slocate naturally uses a database residing on a PSHFS partition. The database was implemented as a persistent hashtable, even though any other similar data structure could be used. To avoid differential effects of disk access, a shared memory partition was used to store both databases, original and PSHFS.

The results shown are averages of 100 repetitions of the measured operations. As seen in Figure 4-5, the difference in time to construct the database is minimal. However, the PSHFS implementation has a decisive advantage in the search operation, which takes an order of magnitude less time on the small test case (100 entries), and more than 2 orders of magnitude less time on the largest testcase (4K entries). As a result, the search time drops from being noticeable (1.8 seconds) to being nearly instantaneous (0.013 seconds).

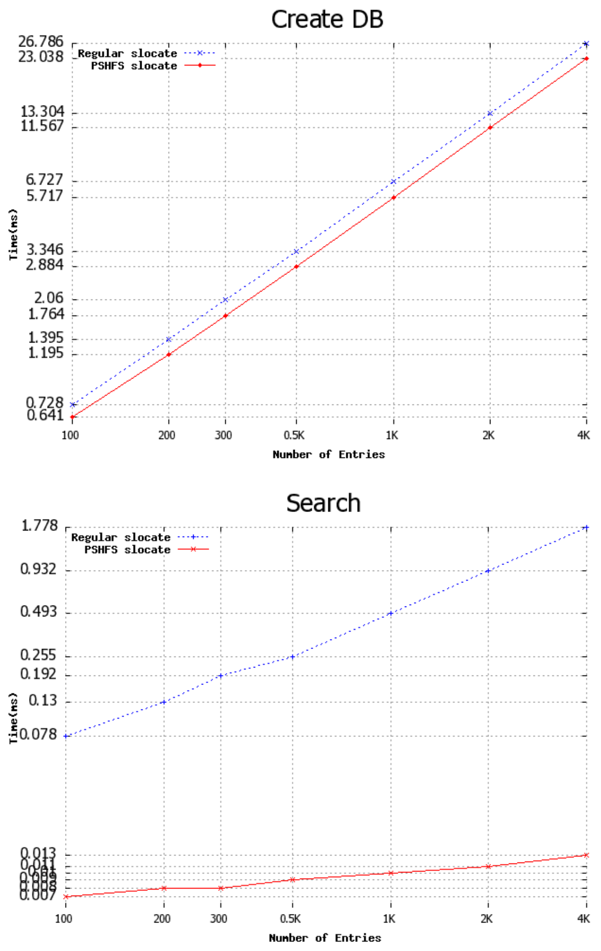

Figure 4-5 results of database creation and file name search using the original slocate and the PSHFS version

5. Related Work

Several projects which share some or even most of our ideas have been proposed over the years.

5.1 Single Level Store

The concept of single-level store, pioneered by Multics in the 1960s and by IBM in the 1970s, and recently used in EROS [Shapiro 2002], regards all storage as a sequence of pages. At any given time, pages may reside either in RAM or on disk. If a process tries to access a non-RAM-resident page, a page fault will occur. This replaces the need for an explicit read. Writes are performed automatically when a modified page is evicted to make space for another. Thus disk space is used only for swap, and there is no explicit file system.

In Multics [multics] all memory was organized as a set of segments, which today can be thought of as being memory-mapped files. The contents of each segment were paged in and out according to usage. This included segments with executable code. Building on this, Multics introduced the idea of dynamic linking, in which a running process could incorporate segments with additional (library) code to its address space. This, in turn, allowed applications to always run with the latest version of the linked code. Segments could also be shared among processes.

Another example of single-level store implementation is the IBM AS/400 [Baldwin 1998], which later became the i5/OS. This is an object-based system, where objects (e.g. a database file or index) are stored in one or more segments of 16MB. Objects are identified by 64-bit virtual addresses, and are naturally paged in or out according to usage. AS/400 applications work only at the object level, and do not need to consider I/O explicitly. An important feature is that objects may be tagged as temporary or persistent.

The AS/400 implementation places each object within a single "auxiliary storage pool" (roughly equivalent to a volume or file system). In i5/OS this approach was changed, with object pages intentionally scattered across all disks so that the objects can be stored and retrieved much more rapidly. The system also allows CPU, memory, and disk resources to be freely substituted for each other at run time to smooth out performance bottlenecks.

The EROS experimental system [Shapiro 2002] also used a single-level store of pages, and the RAM is considered a software-managed cache of this storage space. Like the AS/400, this is an object-based system. Actually there are two types of memory objects: pages and nodes. Pages are pages of user data. Nodes store capabilities and process state. The most important feature of the EROS design is its support for transactional semantics. This is achieved by partitioning the disk space into two: home locations and a checkpointing area. The home locations are the "real" addresses of the objects. But when an object is written, it is first written to the checkpointing area. Data is then copied form the checkpointing area to the home locations only upon achieving a consistent system checkpoint.

Although single-level storage has been available for many years, and provides at least part of our desired functionality (simplified programming and data efficiency), it has not caught on beyond its use in IBM's AS/400. This is probably at least partly due to the relatively coarse-grain granularity of the supported objects. PSHFS attempts to rectify this by supporting a malloc-like API that is naturally used for fine-grained objects, and in fact is essentially the same as the APIs commonly used for dynamic memory allocation today.

5.2 SPHDE

The SPHDE (Shared Persistent Heap/Data Environment) library [Munroe 2007] is the closest we know of to our work. Like PSHFS, it allows processes to allocate memory objects that persist beyond the original process's lifetime, and can be shared by other processes. It demonstrates utility of shared persistent heaps in leveraging large virtual addresses and memory mapped files, and combining memory allocation and file persistence into a single activity. SPHDE demonstrates an improvement in the efficiency of data access and sharing, as multiple programs can access data directly (operate in place) from the single real page copy. This eliminates the need to copy the data through multiple layers of buffering. When all programs share data at the same virtual address, there are also opportunities for the kernel to manage the memory map to avoid aliases and share MMU resources across applications.

To demonstrate the utility of the approach, [Monroe 2007] describes a gigapixel Mandelbrot viewer, which allows users unlimited zoom into a depiction of the Mandelbrot set. This is supported by a quad-tree design, where additional levels of detail are generated on demand and retained for possible future use.

SPHDE is similar to PSHFS in its goals and features. However, it does not provide backward compatibility and current VFS mechanisms can not be used. The "File System" term is dismissed entirely changing the programming methodology. PSHFS is similar in spirit, but also retains the traditional VFS infrastructure.

5.3 Using Flash Memory

Flash is by far the most common non-volatile memory technology, widely used in consumer electronics such as digital cameras and disk-on-key devices. It is therefore natural to consider it as a potential replacement for RAM. However, the current technology is too limited in terms of access speed, usage patterns (e.g. the need to clear a whole block in order to update even one byte), and wear (limited number of write cycles). These restrictive features have led to the design of special algorithms that take them into account [Gal 2005a].

While not (yet) a good replacement for RAM, flash can nevertheless be used instead of disk. The common approach to using Flash memory technology in embedded devices has been to use a pseudo-filesystem on the flash chips to emulate a standard block device and provide wear leveling, and to use a normal file system on top of that emulated block device.

More recently, file systems have been designed explicitly for flash directly [Woodhouse, Gal 2005b]. For example, JFFS is a log-structured file system especially for use on flash devices in embedded systems, which is aware of the restrictions imposed by flash technology and which operates directly on the flash chips, thereby avoiding the inefficiency of having two journaling file systems on top of each other.

PSHFS is inspired by possible use on top of non-volatile memory like flash. However, the current generation of flash devices have various limiting characteristics. Therefore our current implementation is based on volatile memory (RAM) and is not optimized for flash.

6. Conclusions

The PSHFS implementation meets its goals, showing significant performance gains, and a number of other significant advantages over standard virtual memory file systems, including:

- Programming simplification by using a native malloc-like API, when working with shared and persistent large data objects
- Improved efficiency of data access, by using direct pointers rather than system calls
- Elimination of multiple data copies between buffers

At the same time, PSHFS is fully backward compatible, providing a standard file system API.

The PSHFS implementation takes advantage of a unique shared memory file system implementation, adding another layer of abstraction, providing significant performance increase and new methodology of persistent shared storage programming.

Numerous improvements and features are possible to make PSHFS work faster and more efficiently:

- Replace the tmpfs core with a dedicated core, to allow more efficient memory mapping for PSHFS objects.
- Security features have to be implemented to allow a full unix-like security scheme to be applied to objects.
- Compaction mechanism has to be added to prevent fragmentation.
- Use more sophisticated mappings, such as those described in [Itzkovitz 1999], to reduce the page size overhead problem, where a minimum of a whole page is allocated to a file, even if the file size is smaller than a full page.

References

[Baldwin 1998] James R. Baldwin. AS/400 Memory Management, 1998. http://varietysoftworks.com/jbaldwin/Education/single-level_store.html

[Gal 2005a] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138-163, 2005. http://www.tau.ac.il/~stoledo/Bib/Pubs/flash-survey.pdf

[Gal 2005b] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. *Proc.USENIX Tech. Conf.*, pp. 89-104, 2005. http://www.tau.ac.il/~stoledo/Bib/Pubs/usenix2005.pdf

[Itzkovitz 1999] Ayal Itzkovitz and Assaf Schuster. MultiView and Millipage – Fine-Grain Sharing in Page-Based DSMs. *Proc 3rd Symposium on Operating Systems Design and Implementation*, USENIX, 1999. http://usenix.org/events/osdi99/full_papers/itzkovitz/itzkovitz.pdf

[McKusick] Marshall Kirk McKusick, Michael J. Karels, Keith Bostic. A pageable memory based filesystem. *Usenix Summer Conf.* pp. 137-144, 1990. http://docs.freebsd.org/44doc/papers/memfs.pdf

[multics] Multics, WikiPedia, (retrieved Sep 2008) http://en.wikipedia.org/wiki/Multics

[Munroe 2007] Steve Munroe. Exploiting 64-bit Linux. *Linux Journal*, August 1st 2007 http://www.linuxjournal.com/article/9723

[PCM] Phase change memory. WikiPedia (retrieved Sep 2008). http://en.wikipedia.org/wiki/Phase-change_memory

[Shapiro 2002] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the EROS single-level store. *USENIX Annual Technical Conference*, 2002. http://www.eros-os.org/papers/storedesign2002.pdf

[slocate] Secure Locate, 2006. http://slocate.trakker.ca

[Snyder] Peter Snyder. *tmpfs: A Virtual Memory File System*. Sun Microsystems Inc. http://www.solarisinternals.com/si/reading/tmpfs.pdf

[Wilson 1995] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. *Intl. Workshop Memory Management*, Sep 1995.

[Woodhouse] David Woodhouse. *JFFS: The Journaling Flash File System*. Red Hat Inc. http://sources.redhat.com/jffs2/jffs2.pdf