The Design and Implementation of the Creepy File System and the IX File Protocol

Francisco J. Ballesteros
Roger Peppe
draft: 6/7/2012
Laboratorio de Sistemas — Universidad Rey Juan Carlos
`http://lsub.org/who`
Madrid, Spain.

ABSTRACT

Creepy is a file system developed for NIX. It is meant to be fast, simple, and robust, to replace fossil as the main file server at Lsub. Its novelty relies in that it tries to keep most of the file system in memory, using the disk only to store old frozen copies of the main tree, trying to keep all the memory and disk full of data. Permanent archival is not built into the file server program. Instead, external programs can safely archive previously frozen versions of the main tree. Creepy understands a new file system protocol, IX, designed for working on links with poor latency.

1. Creepy principles

Creepy is a new file server program for Nix. The main assumption underlying NIX is that the hardware has changed so much that we should revisit all "well known" facts about system design.

File servers are not an exception, and Creepy tries to apply this principle to file server design. The test machine in use for Nix at Lsub has 64 Gbytes of memory. With that much memory, the machine can easily host in memory everything we use from our main file server. However, our standard file server program insists on using the memory as a cache of the disk, considering the disk as the "master" and mutable copy of the file tree. To make things worse, our standard file server program insist on using the disk as a cache of the permanent archive, so that blocks may be fetched from the archive is not cached.

It is true that many file trees kept on servers will not fit in our large machine. But even so, the mutable part of such trees will fit in memory. Thus, there is no reason to consider the disk as a mutable tree of files.

Instead, Creepy advocates for file servers that keep the main file tree at memory, perhaps the entire tree, perhaps only (the mutable) part of it for huge trees. The disk is used as a tool to survive during machine halts and power outages, and also to keep those parts of the file tree that do not fit in memory.

File servers are used to store data of importance for the user, and they are expected to be reliable. However, they tend to be complex, which harms reliability because of bugs, including programming errors and race conditions.

Much of the complexity of file servers is there to try to make them fast. However, considering the diversity of hardware (including virtual machines, for example), it is not clear if such complexity will actually make them faster on most platforms. To explore an alternative, Creepy tries to achieve good performance by remaining simple and, at the same time, by being aggressive with the use of multiple processes in the implementation.

Creepy tries to keep the file structure as simple as possible (but not simpler), and tries to avoid features that would introduce complexity: archival, support for mirroring and stripping, transaction support, etc. Instead, many such features (archival, mirroring, stripping, etc.) are to be provided by external programs. The file system program has just to make it feasible, not to take over all related tools.

It also tries to use multiple processes (many, actually) both to structure the program and to make it faster.

Unlike UNIX file servers, Creepy is implemented at user-level, not within the kernel, and it is designed for machines that serve files through the network. Thus, to compare its performance with respect to UNIX it should be compared with network file servers, and not with local in-kernel file systems. That said, Creepy can be used for local storage as well if that is required.

In what follows we describe not just the current design and implementation, but also the ones previously used in order to document the design process and the experience learned.

2. Interface

The interface for the file server is the file tree exported. There is no separate configuration interface. Instead, the configuration console and any configuration file is provided as part of the tree exported.

2.1. Mark II

This implementation provides a tree with the following layout:

/ The root directory is never found on disk. It is a placeholder for the contained files.

/cons is a synthesized file used as a console for file system administration, and it is not found on disk.

/stats Another synthesized file used to report statistics.

/root The root of the file system, where user files are kept.

Mounting the file system with an empty tree name yields the tree rooted at *root*. Using *main* yields the tree as shown, to gain access to the console and other files.

All files including the console and statistics are subject to file permissions. This permits using them for securing file system administration and profiling. Because they are exported through the same file protocol used to export the actual file tree, they are available at any client without requiring different access means to the machine running the program.

2.2. Mark I

This is the interface used in the first prototype. It exports a file tree with the following layout:

```
/
/active
/cons
/stats
/archive
/archive/1331321206581312000
/archive/1331321207419554000
 ...
```

The main difference is that it provides access to roots for frozen trees (found on disk) not yet reclaimed as free space. Doing so implied that the file tree was not a tree, but a DAG. Note that archives are temporary. Permanent archives were created by copying them into an archival facility, once per day.

As a result, locking and free space reclamation got more complex. Mark II simplified such assumptions by removing access to old versions of the tree. There are no temporary snapshots in Mark II. For permanent archival, a frozen tree is read from disk and then archived.

3. Operation

3.1. Mark II

The mutable file tree is kept in memory. Memory is not considered just a cache, but the only mutable tree. Parts of the tree might be left on disk (thus be frozen) if it does not fit in memory.

On a periodic basis, and also when more than a configured number of dirty blocks is reached, the tree is frozen and written on disk. The disk is considered a log, with new blocks written to the end.

When the disk gets full, a mark and sweep process marks all reachable blocks and deems all other ones as free. Thus, the log is winded around the partition and jumps to the next free block used, instead of been a contiguous circular log.

Consistency on disk is achieved by writing all blocks in any order, and then writing a new superblock referring to the new */root* as kept on disk. Failure yields the previously written tree. Success yields a new and consistent tree.

3.2. Mark I

The initial implementation is more complex. It relies on reference counters to maintain a DAG of blocks and detect unused ones. But blocks are still WORM and, once written, are never changed (although they might be released and then reused).

Consistency is achieved by using two sets of reference counters. The super block makes one of them active, and toggles between them on each write. Thus, if writing a tree fails, the counters for the old tree are in effect, and newly written blocks are simply ignored.

Initialization of the partition happens on demand, which means that there is no need to zero-out the entire disk before using it. Also, structures and sizes are logical and do not follow hardware block sizes. The partition may be extended (or copied) if the existing data is preserved (or copied verbatim).

4. Disk Structures

4.1. Mark II

The disk is structured as shown in figure 1. The file system is made out of blocks, of a fixed size. Usually, 8 Kbytes. The first block is not used. The second block is used to keep the super block. The rest of the disk is an array of block-groups. A block group has as many blocks as 32-bit words fit in a data block (the last one might have less blocks depending on the disk size), because a tag is 32 bits. Each block group starts with a *DBtag* block followed by *DBdata* blocks.

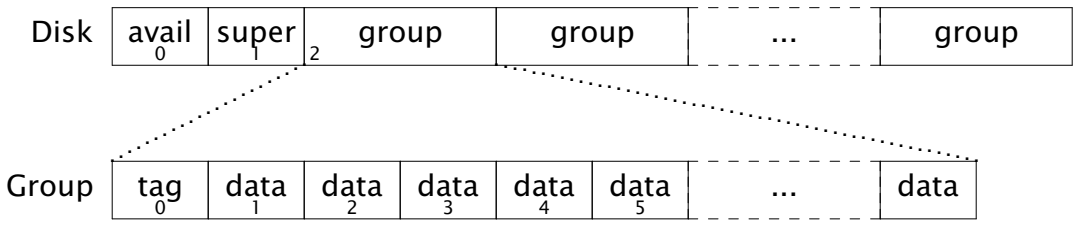

Figure 1 Disk structure. Numbers correspond to full blocks.

The *DBtag* block keeps the tags (or marks) used during the mark and sweep process. A tag is a 32 bit number, described later.

These are the block types and well-known addresses and sizes. The types for file, indirect, and direct blocks permit arithmetic to obtain the type of next blocks in chain during file traversals. Blocks for directories can be operated in the same way, only that they always have *DBdirflag* set. This makes it easy to know when a data block has directory entries and when does it have user data.

```
enum
{
    DBdirflag = 0x100,

    /* block types */
    DBfree = 0,         /* free block */
    DBattr,             /* unused */
    DBsuper,            /* super block */
    DBtag,              /* tag block */
    DBfile,             /* file */
    DBdata,             /* data block */
    DBptr0 = DBdata+1,  /* simple-indirect block */
                        /* double */
                        /* triple */
                        /*...*/
    DBdir = DBfile+DBdirflag,   /* dir */
    DBdirdata,                  /* dir data block */
    DBdirptr0 = DBdirdata+1,    /* dir simple indirect */
    /* ... */

    Daddrsz = BIT64SZ,
    Dtagsz = BIT32SZ,
    Dsuperaddr = Dblksz,
    Dblk0addr = Dsuperaddr+Dblksz,
};
```

All disk addresses are 64-bit words. All integer values stored on disk are kept in little-

endian format. All strings known by the file server program are kept encoded in UTF-8, including the final Nul byte. The program assumes a little endian machine, but is prepared to be ported to a big endian system: blocks are translated to/from the machine format from/to the disk format while reading or writing them. Currently, these two functions simply check that the machine is indeed little endian (this is a bug), but they can easily convert the integers to permit creepy to run on big-endian machines.

DBfree is only used to refer to memory blocks recycled on the free list, and is never explicitly used on disk. *DBsuper* is the super-block, *DBfile* describes a file, *DBattr* is used to implement large file attributes (but it's not yet supported), *DBdata* contains raw file data (or directory entries for directories), and *DBptr* blocks contain addresses of other blocks. Directories are files that contain an array of disk addresses as data (one per file), free entries are zeroed and ignored during reads.

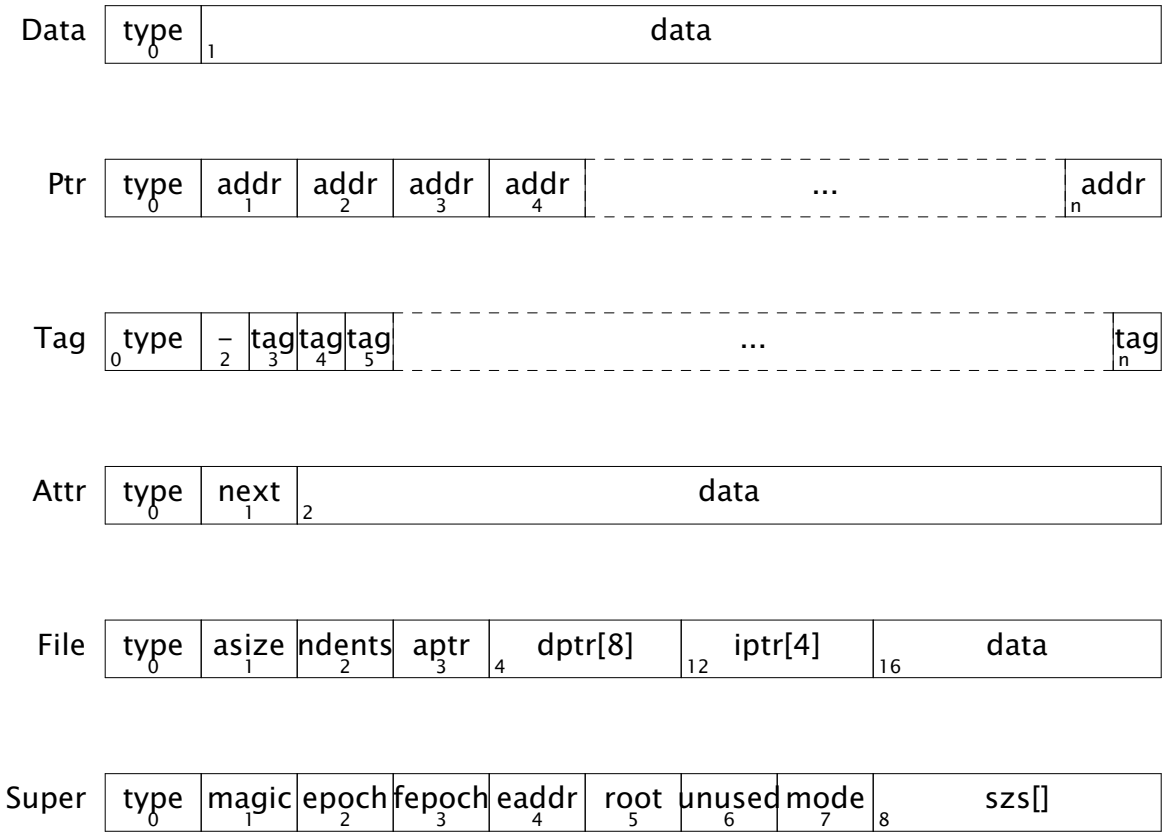

Figure 2 Block types. Numbers indicate the word number (64-bit words). Blocks are usually 16 Kbytes. All disk addresses are 64-bit words, but tags are 32bit epochs. All integer values stored on disk are kept in little-endian format. All strings known by the file server program are kept encoded in UTF-8, including the final Nul byte.

All blocks start with the block type, for safety checks. Other fields should be easy to understand provided what we say later about the different blocks.

The superblock describes the file system. The root of the tree kept on disk (corresponding to */root* on the served tree) is addressed by *root* on the super block. Fields *epoch* and *fepoch* are used in the mark and sweep process. Field *eaddr* is used when operating in WORM mode, which nevers collect blocks and uses an actual log. The

operating mode is described in *mode*. All other fields in the super block contain sizes for data structures (e.g., block and block group sizes) as set at initialization time, to double check that the program using the disk can actually understand it.

It is not shown in the figure, but the super block contains two different copies of its data: the official copy and a second copy kept at half the block size. Both copies must match. If they don't match, it must be because of a failure while writing the super block. In that case, the oldest copy is used, usually discarding the last frozen version of the tree.

A file on disk is identified by the address of a *DBfile* block. Such block defines both data and metadata for a file. Metadata is stored at the start of the embedded data area within the *DBfile* block, perhaps extended with more data stored in *DBattr* blocks (not yet supported), linked on a list starting with the block addressed by *aptr*. The size of the metadata is identified by *asize*. The rest of the embedded data area is used for the initial data of the file. For short files, all file metadata and data is kept in a single block.

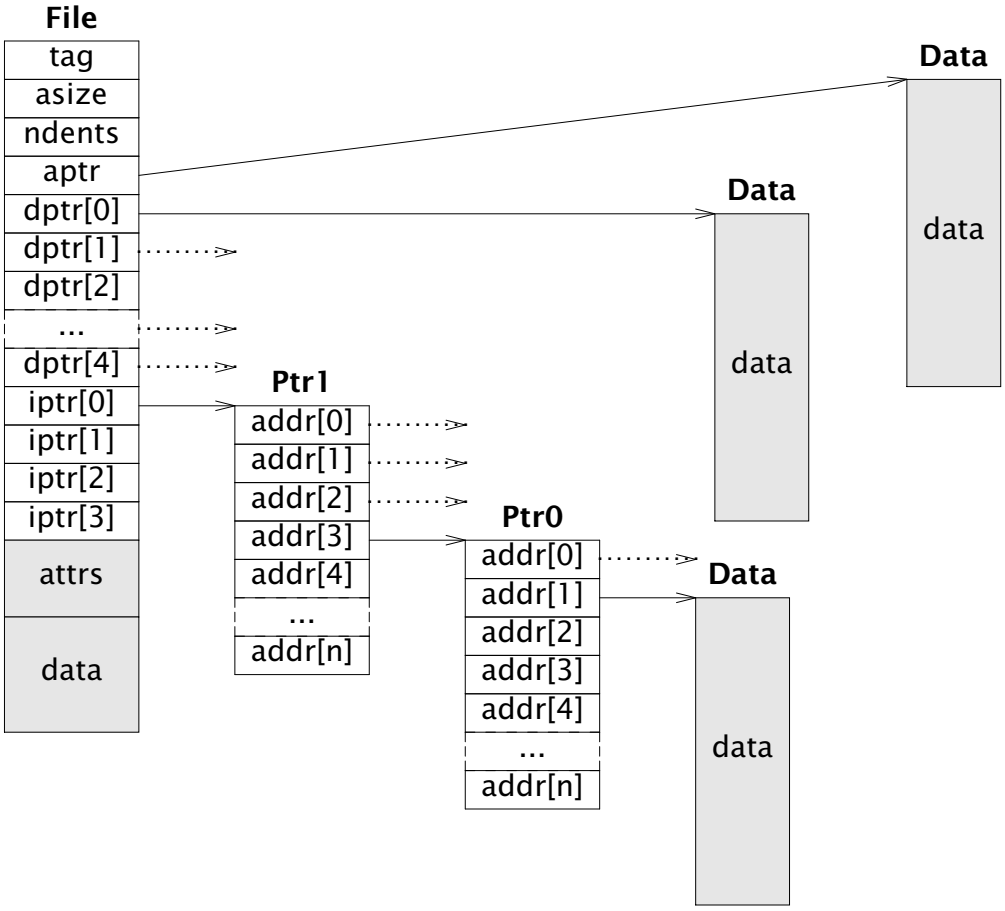

Figure 3 File structure on disk. Only some of the blocks are shown. All data areas are shown filled. The data portion within the file block is used both to keep metadata (plus some room for metadata growth) and the start of file data. Data that does not fit within the file block is allocated on data blocks rearched from pointers kept in the file block.

There are n direct data pointers and m indirect data pointers. The indirect pointers require *n*-indirections to reach data block, being *n* the index on the indirect array plus one; e.g., *iptr*[0] is a single-indirect pointer, *iptr*[1] is a double-indirect pointer, etc. The type for the block addressed by *iptr*[n] is *DBptr*n, although all such blocks keep the same layout (an array of disk addresses, preceded by the block type). Mark II uses 8 direct pointers and 4 indirect pointers.

File metadata consists on a series of *name* and *value* pairs, with some of them always present for all files. In the current implementation, only the predefined attributes are supported, and they are stored on disk as implied by this definition:

```
struct Dmeta            /* mandatory metadata */
{
    u64int  id;         /* ctime, actually */
    u64int  prev;       /* address of previous version, hint */
    u64int  mode;
    u64int  atime;
    u64int  mtime;
    u64int  length;
    u64int  uid;
    u64int  gid;
    u64int  muid;
    /* filename\0 */
};
```

These are to be followed by further names and values. Names are to be stored in UTF-8, and values by storing the value length (32-bits) followed by the value data. By convention, integer values must be 64 bits, string values must be stored in UTF-8, including a final Nul byte (also counted in the value length).

Within the program, user identifiers are 64bit integers. However, they are strings in the protocols spoken. The users file, kept at */active/users*, maps internal user identifiers to user names as seen externally. Measures are taken to ensure that known users are never forgotten, and that the mapping between users and identifiers is preserved, as described later.

When speaking IX as the file protocol, all attribute values (including numeric ones) are encoded as strings in UTF-8.

The predefined attribute ''*'' is not stored on disk. It has as its value the set of names (separated by spaces) for all attributes present on a file.

Directories are implemented as regular files. Their data is an array of disk addresses, one per contained file. The array may contain holes when files are removed, but the *ndents* field in the *DBfile* block records the exact number of contained files. This is used to iterate over the different children files, ignoring null entries, and detect the point when we have found all entries of interest. Also, this makes it easy to check for empty directories during file removal.

4.2. Mark I

In this implementation, now deprecated, each block group starts with two *DBref* blocks followed by *DBdata* blocks. All blocks start with a *tag* encoding both the block type and address, for error checking.

A *DBref* block holds the reference counters for all the blocks in its group (and thus, the first two entries are not used). Only one of the two *DBref* blocks is used at a time. The field *odd* in the *DBsuper* block selects even or odd reference blocks. This is used to

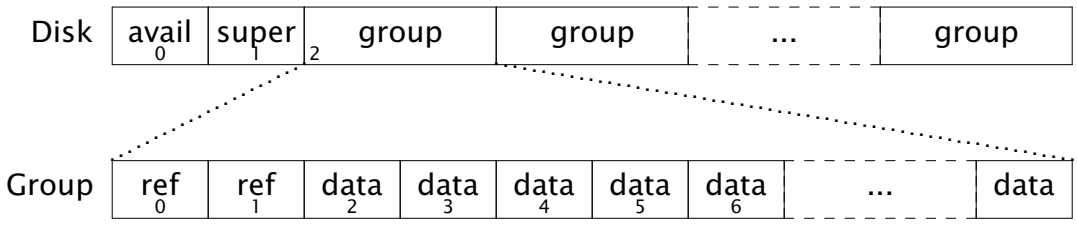

Figure 4 Disk structure. Numbers correspond to full blocks.

ensure that the set of reference blocks is coherent despite failures. *DBref* blocks also double to support a free list of blocks.

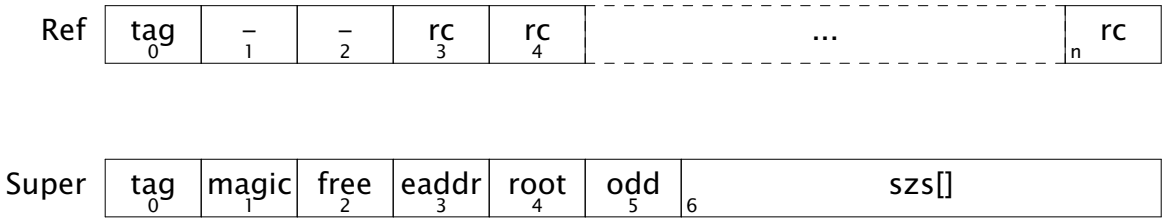

Figure 5 Block types used in Mark I. All other block types are similar to those in Mark II.

For allocation, *eaddr* in the super block identifies the end-address for the already allocated portion of the disk. When all blocks before that address are in use, the address is incremented in multiples of block-size to allocate more blocks (until the disk gets full).

Besides the portion from *eaddr* to the end of the disk, there may be free blocks as identified by a free list. The first block in the free list is addressed by *free* in the super block. Because the reference counter for such block ought to be zero, its reference counter (in the corresponding *DBref* block) is used to provide the address of the next free block on disk, or to signal the end of the free list by storing a zero. Note that all the blocks on this list have a zero-valued reference counter (although their counters are not zero in practice).

Most bugs found in Mark I came from the free list and reference counters. Both are gone.

5. Utilities

5.1. Workers

Processes created to serve protocol requests and to write blocks to disk are provided by the *worker* library, taken from Plan B. This library maintains a pool of workers that grows as required to satisfy the work load, and keeps idle workers cached for future work.

There is a configured parameter in Creepy that places a limit on the number of outstanding RPCs per client (i.e., per connection). When such limit is reached, further requests are served directly by the process reading protocol messages, instead of being handed to a worker. Another limit operates in a similar way with respect to the number

of processes used to write blocks to disk.

5.2. Errors

Error handling is provided by the *error* library, with an interface similar to that used in the kernel, but adapted for use both with and without the thread library. This makes it easier to release references and resources upon errors.

A significant change is that the *error* function accepts arguments similar to *print*, including the *%r* verb. Thus, functions may decorate error messages as the stack is unwinded.

5.3. Allocators

Most structures are frequently allocated and released. To prevent fragmentation, most of them are allocated from allocators, defined as:

```
struct Next
{
    Next *next;
};

struct Alloc
{
    QLock;
    Next *free;
    ulong nfree;
    ulong nalloc;
    usize elsz;
    int zeroing;
    int fixedsz;
};
```

Many of these allocators grow one element at a time, keeping all of the released ones cached. These have zero in *fixedsz*. Some of them (e.g., the memory block cache) have a fixed number of entries, as indicated by a non-zero *fixedsz*. Depending on *zeroing*, the memory of the elements may be zeroed before given to the user.

6. Memory Structures

6.1. Allocators

These are the allocators in use in Mark II:

mballoc Memory block allocator.

mfalloc Allocator for on-memory file information or *Mfile*.

fidalloc Allocator for *Fid* structures used for both 9P and IX.

rpcalloc Large RPC allocator.

srpcalloc Small RPC allocator.

clialloc Per-client information allocator.

pathalloc Allocator of *Path* structures used by fids to refer to files.

The large RPC allocator is used for 9P requests and for IX requests that carry user data. The small RPC allocator is used for IX requests that do not carry data, consisting on just a few bytes.

6.2. Blocks

Most of the memory is spent in a huge allocator used to keep blocks in memory. A block on memory is represented by a *Memblk* structure, which among other things like locks and pointers for lists, contains a verbatim image for the corresponding disk block, or *Diskblk*. In what follows, whenever we refer to a block loaded in memory, it should be clear that it is a *Memblk* and that the memory address for the block is a pointer to it, not to be confused with the disk address for the block.

```
enum
{
    MBfree = 0,
    MBmem,
    MBout,
    MBclean,
    MBin,
    MBlru,
    MBerr,
};

struct Meminfo
{
    Memblk  *next;      /* in hash or free list */
    Link;               /* clean, out, ... lists */
    Ref;

    int type;           /* block type */
    int state;          /* Mfree, Mnew, ... */
    daddrt  addr;       /* block address | 0 */

    QLock   slk;        /* state change */
    QLock   ldlk;       /* wait while loading */

    Mfile   *mf;        /* DBfile on-memory info. */
    Channel *wc;        /* for write super */
};

struct Memblk
{
    Meminfo;
    Diskblk d;
};
```

A block may be at any of these states:

MBfree Unused block.

MBmem Block with on-memory address, mutable and not yet written. Not linked at any list.

MBout Block given a on-disk address, frozen and not yet written. Linked at the *out* list.

MBclean Block written to disk (addressed by a disk address) or read from disk, which is frozen. Linked at the *clean* list.

MBin Block given a on-disk address, being read and not yet ready for use. Not linked at any list.

MBlru Block being removed from the memory cache. Not linked at any list.

MBerr Block with I/O errors while reading or writing it. Not linked at any list.

Transitions are as shown in the figure.

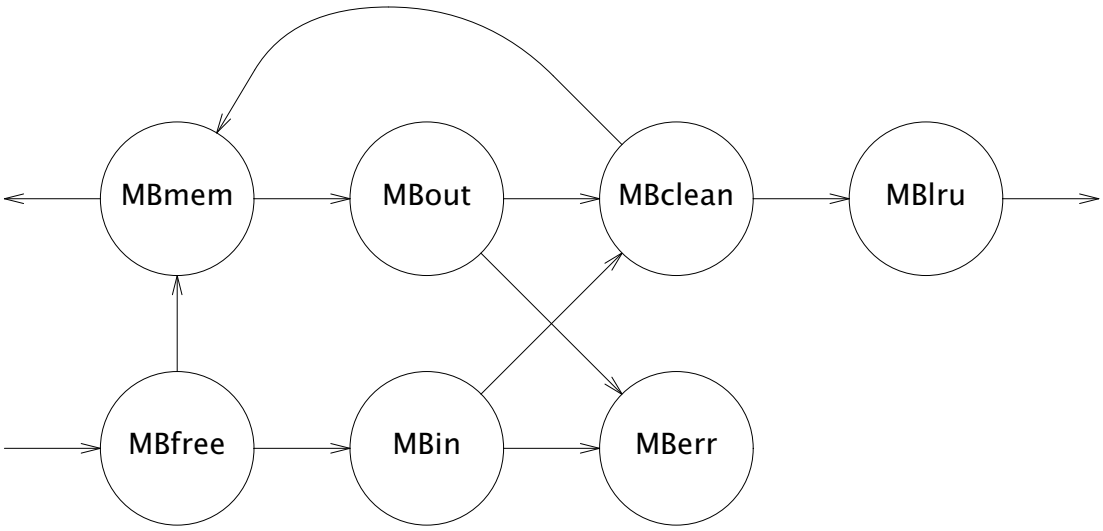

Figure 6 Memory block state diagram.

When blocks are given memory addresses they are hashed by their memory address (the pointer to the block structure). When they have disk addresses they are hashed instead by the disk address. The same structure (and its disk block buffer) is used in-place during state transitions, no actual copy on write is performed.

Clearly, memory addressed blocks may refer to disk addressed blocks, but, disk addressed blocks may not refer to memory addressed blocks. This invariant forces an order in the process of freezing the file tree for writing it on disk, and also on the process of melting it (or renewing it) for making changes on it. Such ordering is consistent with the lock ordering: parent to child.

Blocks created on-memory for new data start on *MBmem*. If they are removed before writing to disk, they are simply released (become *MBfree*, indeed). When they are written, they proceed through *MBout* and *MBclean*. They can stay there and be discarded due to LRU or they may be ''renewed'' to be once more *MBmem*, for further updates. If a frozen block is to be renewed, the block is moved ahead in the *out* list (so it is written soon) and the implementation waits until the block becomes *MBclean*. At that point, the address is changed (and the block re-hashed) to the memory address. The old on-disk address is forgotten. The next time the block is written it will acquire a new disk address.

While on memory, blocks are reference counted to know when they may be released to the free list. The links from the hash, clean, and dirty lists collectively account as a single reference to each block. That is, unhashing a block will make it be released as soon as others referencing the block go away. However, blocks kept in the hash (and clean or out lists) are kept even when no other reference is kept. The figure gives an overview of how blocks are kept and linked in memory.

DBfile blocks maintain extra on-memory information for the file, represented by an *Mfile* structure:

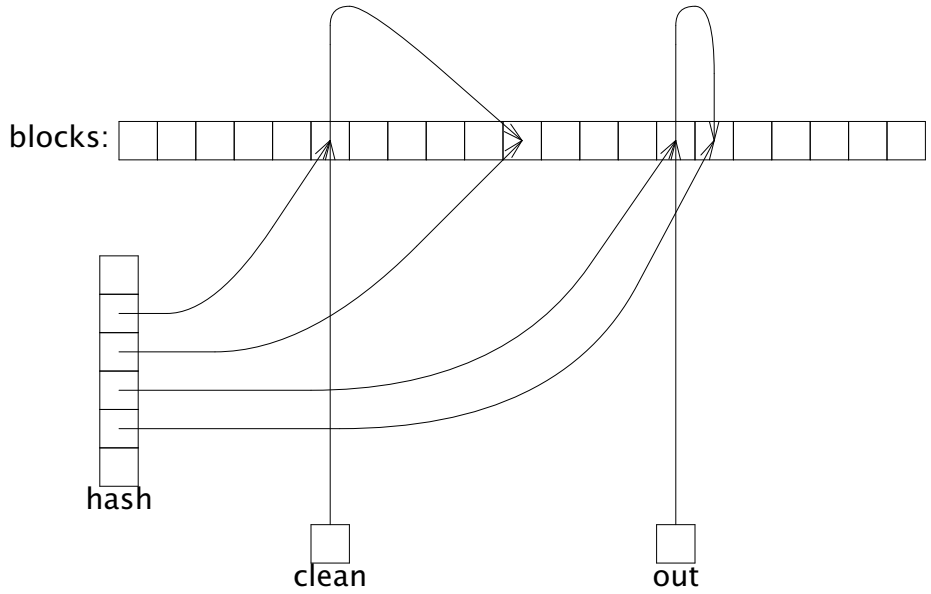

Figure 7 Memory structures. Blocks are kept in an array of blocks allocated at start-up time. They may be retrieved by disk or by memory address (depending on their state) using a hash. In certain states, blocks are also linked on a list keeping all blocks in that state.

```
struct Mfile
{
    Mfile*  next;       /* in free list */
    RWLock;

    char    *uid;       /* reference to the user table */
    char    *gid;       /* reference to the user table */
    char    *muid;      /* reference to the user table */
    char    *name;      /* reference to the disk block */

    ulong   lastbno;    /* last accessed block nb within this file */
    ulong   sequential; /* access has been sequential */

    int open;           /* for DMEXCL */
    int users;          /* is this /root/users? */
    uvlong  raoffset;   /* we did read ahead up to this offset */
};
```

Such extra information includes the unpacked metadata, information for read ahead, a per-file read/write lock, and a pointer to the possible melted version for this file, if frozen.

Memory structures do not keep references (nor pointers) to file parents and children. To navigate the file hierarchy, addresses of children (as kept in data areas for directories) are used to fetch the corresponding (DBfile) blocks.

6.2.1. Mark I

Mark I operated almost like Mark II regarding blocks, but it relied on copy on write to melt frozen blocks, and had to advance references from *Fids* to files to try to make them refer always to the most recent melted version.

In Mark I a pointer is kept between different memory blocks is the *melted* pointer from a *DBfile* to its next version. It is used to advance file references from the user to the melted versions whenever possible, to try to keep the frozen trees unused so they could be reclaiming, and to make sure that the user gets the newest version for a file. This could have been changed to permit binaries in use to stay frozen despite writes to their files, by not advancing the references to files that are open only for reading and are not append-only.

6.3. Paths

The file server program is designed for operation under both 9P and IX. In both cases, each file reference kept by the client, or *Fid*, maintains a *Path* to refer to the file in use. The *Path* structure keeps references for all the files (i.e., for the *DBfile* memory blocks) in the path of the file pointed to by the *Fid*. A simplified view is depicted in the figure. In reality, the array of references is dynamically resized and *Paths* have fields to be linked in their allocator (for reuse). Also, they are shared (with reference counters) and copied on write when needed, as a result of cloning and walking *Fids*.

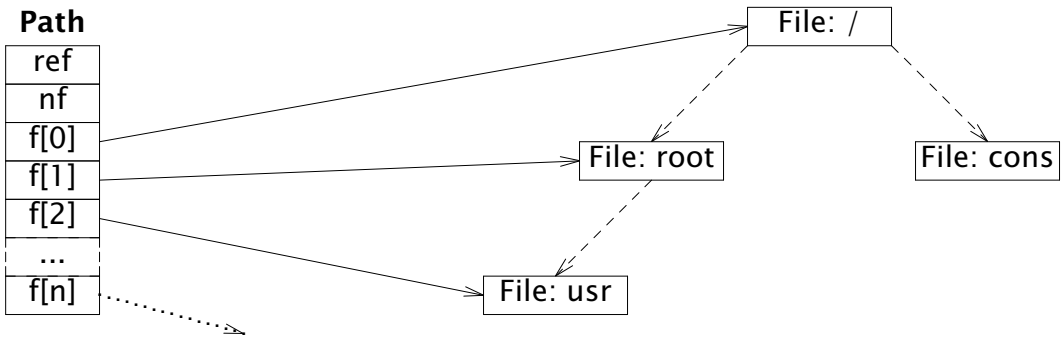

Figure 8 The Path structure is used to refer to a file in use by the protocol client. It keeps one reference (counted as such) to each file in the path for the file of interest.

A consequence is that referenced files are never removed from memory, because they keep at least one extra reference per path. So are their ancestors.

Another useful feature enabled by *Paths* is that walks to the parent of a file are performed without accesing the actual file tree, by discarding the last entry. This was important in Mark I since files may be shared in different trees because of freezes and melts, yet the user has walked only a particular path to reach the file. Mark II makes this irrelevant.

The walk of synthesized names for archived trees in Mark I, using the time in a symbolic format, works well because of the same reason. Otherwise, walking up after entering, e.g. */archive/15:30*, would not work properly.

The main utility of *Paths* in Mark II is that locking can proceed from parent to child to renew files that we must write.

7. Disk allocation

7.1. Mark II

Allocation in Mark II comes from realizing that the debug checking code in Mark I, implemented to confirm the results of the production code, was fast enough for actual usage and was a lot simpler. The code was removed and the debug code was promoted to become the actual implementation. These are the elements used:

- *DBtag* blocks contain an epoch tag for each block.
- The super block contains an *epoch* field to describe the current tag and a *fepoch* field to describe gone tags, now considered free. The current tag, or *epoch*, is always greater than *fepoch*.

Initially all tags are zero, which means their respective blocks are free. During operation, all disk block allocated are tagged with the current *epoch*. (Allocating a block is simply giving an address to a memory block, which might acquire different disk addresses during its life).

Releasing a block is easy: we do nothing. Thus, an entire hierarchy of blocks may be released just by zeroing the reference to its root, wherever it is.

When there are no free blocks (i.e., scanning for zero tags fails), the reachable tree is tagged with a new epoch, and *fepoch* is advanced to the old tag. After all tags and the super block has been written, the old unused blocks are gone.

The sweep process proceeds by locating old tags and zeroing them out, so they could be allocated again.

System activity may proceed during mark and sweep. As a result, marking requires tagging all blocks that can be reached from either the on-memory root or the on-disk root. Other than this detail, the implementation is trivial. It relies on the per-file locks to synchronize with user activity.

7.2. Mark I

Disk blocks are reference counted. When their reference count gets down to zero they are considered free (and linked into a list of free disk blocks using the reference counters to store the next pointers). Of course, this does not apply to the super block or to *DBref* blocks, which are always kept in use.

On disk, the reference to the olddest frozen tree is dropped only when a low water mark is reached. This is a cascade process because references from that block are dropped as a result. The process continues until reaching a high water mark for disk space.

Most bugs in Mark I came from this part of the implementation, and it made normal operation slower.

8. Policy

8.1. Mark II

The policy determines when to evict blocks from memory, when to flush changes to disk, and when to scan for free blocks (or even mark and sweep). In mark II, the policy sends a message to any or both of the LRU and sweep processes, to release memory blocks and/or to collect free disk blocks.

Such messages are sent before each RPC, non-blocking if we are above a water mark, and blocking if we are below. Most of the complexity of Mark I in this respect is gone, because the simplification in the block management made it easier to perform the different activities concurrently without cascading effects.

8.2. Mark I

From time to time the state of the file tree is frozen as described later. Frozen blocks are written to disk and then moved to the clean block list, where they might be used for replacement if the entire file tree does not fit in memory. This happens when the percentage of dirty blocks reaches a water mark, and also on a periodic basis, to prevent loss due to power failures and system faults.

These conditions are checked by a *fspolicy* function called after each RPC, to prevent adding latency to user requests. Activity continues while the policy runs. Thus, there are zero water marks for space in memory and disk, checked before each RPC, to be sure that there are enough resources to serve it. When that is not the case, the policy is called before the RPC. This should happen only when the policy (called after another RPC) does not have time to release resources during intensive usage of the file tree. Water marks are adjusted so that the zero water mark is never (or seldom) reached.

Things are a little bit more complex because being low on memory may require the tree to be frozen, if there are many dirty blocks:

1. If we are low on memory, *fslru* is called to reclaim clean blocks, and if are still low on blocks after this, we pretend we are high on dirty blocks.
2. If we are low on disk space, we call *fsreclaim* to drop old frozen trees.
3 If we are low on disk, or high on dirty blocks, or it was long ago the last write of the file system, we call *fssync* to write dirty blocks.
4 Finally, if we were low on memory or high on dirty blocks, we call again *fslru* to release blocks that might be clean now.

This is not optimal, but it is the best combination we found in usage experiments. More adjustment is needed once we see how the system performs while in production.

9. Mark I Time Walks

Time walks are gone in Mark II. They were implemented in Mark I to navigate The*/archive*. allocation and deallocation.

The */archive* directory provides support for time walks in Mark I, similar to Plan 9's file servers. The names of directories contained under */archive* correspond to the time in nano seconds as provided by *nsec*. This is not amenable for humans, but it is convenient for programs.

As an aid, */archive* accepts other names in *walk* requests. In particular, these are understood and honored:

- yyyymmddhhmm
- yyyymmdd
- mmdd
- hh:mm

The result of the walk is the last directory not newest than the time implied by the walked name. That is, the tree as of the given time.

There is another interface for performing time walks, for cases when we what to walk to exactly a given directory, but in its archived form. Any directory accepts walks to the following names:

- @yyyymmddhhmm
- @yyyymmdd
- @mmdd
- @hh:mm

The meaning is as explained before. However, walking this name leads to the archived version of the directory where the walk is performed. Permissions are still checked, as if the user walked through the entire path from */archive*. Also, it is not feasible to perform a time walk while in a time walk. To walk to another time the user must back-up, leading to the directory in the active tree where the walk started.

The implementation for this type of time walk adds a new entry to the *Path* used by the *Fid*, corresponding to the resulting (archived) directory. A further walk to the parent directory simply removes the last entry from the *Path*. Therefore, things are kept consistent as far as the user can see.

10. Archival

10.1. Mark II

Archival in Mark II does not even require a different program, although there is almost no support for permanent file archiving.

The archival program, *rip*, is exactly the file server program, *9pix*, started in WORM mode. In this mode, no mark and sweep process will ever happen. The drawback with respect to fossil is that there is no file de-duplication, like in Venti. The benefit is that the program is a lot simpler and faster. Considering the number of Venti arenas we are using and the ones we have available, we are willing to sacrifice de-duplication for reliability and speed.

Arch, the archive program is actually, once more, the file server program. Instead of its *main* routine, it uses a different one that retrieves the tree, compares with the last archive, and creates a new one. It is started by *9pix* when archival is configured, with a pipe connected to its standard input that is used as described next.

Arch reads blocks by asking them (by address) to the main file server, through a pipe. That way it leverages the cache kept in the memory of the file server and, at the same time, can retrieve the stable tree kept on disk without any interference to normal file server operation. Blocks asked that are not in the cache are discarded once served, without polluting the LRU resulting from actual file server usage.

Rip cooperates to maintain archives by supporting a *link* control request (in the console), enabled only in WORM mode. *Arch* starts by locating the last archive and linking it to a new name, using the customary year/month-day paths. Because the archive is frozen (they are clean blocks), any change made to it leads to a copy on write of disk blocks (but memory blocks are reused and not copied).

Besides implementing the block read protocol through a pipe to *arch*, *9pix* cooperates by propagating modification times up to the root each time a file is modified. Thus, locating the subtree that has changed since the last archive is faster, because unchanged parts of the tree can be pruned without reading the entire tree metadata.

10.2. Mark I

Mark I considers *venti* for archival of daily trees, but does not include support for archival in *venti*. External programs like *vac* and *vacfs* are to be used for archiving a frozen tree and for navigating the archive. This can be done without races because all tress found under */archive* are frozen, and thus quiescent. The policy function reclaims old frozen trees starting from the oldest one, and refuses to reclaim the last frozen tree. Therefore, there is always at least one frozen tree.

As long as there is disk space to keep a frozen tree, plus some space for future freezes done while copying the tree in *venti*, there should be no problem.

Vacfs has to be modified to honor *users* regarding groups and permission checks, but other than that, the functionality provided is equivalent to that provided by the archived tree in *fossil*, without direct support in Creepy for such feature.

11. IX

It is common to use Plan 9 file servers through slow network connections. Tools like CFS, OP, and others try to help there. But it would be desirable to be able to work from remote terminals suffering poor-latency links, keeping coherency with a main file server when feasible. The IX file protocol has been designed for NIX with that in mind.

11.1. IX design

IX is built by leveraging what we learned from Op, the existing code-base for 9P, and ideas as discussed previously in the community [.streams 9p.]. The protocol is built upon a few design guidelines:

- The underlying transport is a reliable, ordered, connection—similar to a TCP stream—, with flow control for network congestion.
- An RPC in the protocol is a series of elementary transactions operating on a single file. Such transactions are similar to 9P requests.

IX is built upon the concept of connection channels. It is extremely cheap to build or dismantle a channel, so that it is reasonable to create one for each RPC. Channels are duplex, and inherit the reliability and ordering properties from the underlying transport. A single connection is multiplexed among multiple channels so that there is no starvation for sending or receiving through them.

An IX client would create a new RPC by just creating a channel. This does not require communication with the peer, and is a local operation. Then, one or more transaction requests will be sent through the new channel. The last request is flagged as such; thereafter no more requests may be written on the channel. Only the client can allocate new channel identifiers, which are local to the connection being multiplexed.

An IX server receives requests through allocated channels and processes sequentially all requests for a given channel. It ceases to process them when one fails or when it finds the last request, whichever happens first

Note that channels are different from 9P tags. Like tags, they identify a particular outstanding RPC, so that multiple RPCs may be in transit at the same time. But, unlike tags, channels permit huge amounts of data to be sent (concurrently with requests for other channels) and each direction in the duplex stream can be closed independently.

Once a client has sent the desired transaction requests through a channel (or perhaps concurrently with them), a client receives through that channel individual replies for all transactions sent. An error reply to a transaction indicates that the RPC is

finished, and the channel is closed.

To identify files, IX relies on *fids* similar to those in 9P. But, unlike in 9P, the server defines which values are to be used for new fids. The client has to keep its own data structures for files, which means that it has no advantage by selecting fid numbers. On the other hand, the server might exploit fid values to improve the data structure used to keep and look up fids.

File metadata is a set of name and value attributes, with some attributes predefined. Currently, the predefined attributes match those implemented by Creepy. The *id* thus replaces the notion of a *Qid.path* in 9P; and the *mtime* is used instead of a *Qid.vers*. The file type is kept encoded in the mode bits for the file, as it is in Plan 9. There is no *Qid.type*.

Because a channel implies a context for individual transactions, it is feasible to simplify 9P transactions for use in IX to avoid unnecessary duplication through the wire. For example, once a fid has been established for an RPC, it is not necessary to repeat its value for each following transaction. The set of simplifications made is described in the next section.

To support aggressive caching, a conditional transaction has been added to the set of 9P requests known by IX. This transaction, *Tcond*, asks the server if a piece of metadata for a file is the same, greater than, less than, or different than the given value. The same relational operation can be performed for multiple elements of the *stat* information for a given file. If the condition holds, the server replies with an *Rcond* reply. Otherwise, the server replies with an error indication, thus terminating the RPC.

11.2. IX Operations

An RPC in IX is a series of IX operations sent through the same channel. All messages are sent using this format:

```
len[2] msg[len]
```

Two bytes encode the message length, followed by that many bytes of data. Note that this places a limit only on an individual operation, not on the entire RPC.

Expanding the bytes of a message, we now see:

```
len[2] chan[2] type[1] ...
```

The *chan* identifies the channel, including one bit to flag the first message through a channel (causing its allocation) and a bit to flag the last message through a channel (in that direction). Both bits may be set in a single message. A channel is deallocated when both directions have seen a message flagged as the last one. Error replies always are flagged as the last message sent. The *type* field identifies the message type.

These are the operations defined in the current version of the protocol, as implemented for Creepy:

```
Tversion msize[4] version[s]
Rversion msize[4] version[s]
```

Negotiates the limit on the maximum message size and the protocol version.

```
Tattach uname[s] aname[s]
Rattach fid[4]              // fid becomes the current fid
```

Attaches the user *uname* to the root of the file tree *aname* and provides a *fid* for the root of the tree to the client. This *fid* is the one to use for further operations in the

channel. Also, authentication had to happen before speaking IX through the channel, although this bit may be subject to change in the near future.

```
Tfid fid[4]
Rfid
```

Sets *fid* as the one to be used in further operations through the same channel. This request may be used multiple times if the same RPC has to refer to different fids.

```
Rerror ename[s]
```

Responds to a transaction with an error message. It is always flagged as the last message in the channel (from the server to the client).

```
Tclone cflags[1]
Rclone fid[4]
```

Creates a clone of the implied *fid* and informs in the reply to the client of the new *fid* number (the clone). The request includes flags to ask for the clone to be closed on errors and/or closed on the end of the conversation. After this request, the implied fid is the one just created.

```
Twalk wname[s]
Rwalk
```

Walks to the given name, which is a single element in a path.

```
Topen mode[1]
Ropen
```

Opens the implied fid according to the given mode.

```
Tcreate name[s] perm[4] mode[1]
Rcreate
```

Creates a file with the given name and permissions. The implied fid is walked implicitly to the new file and open according to the given mode.

```
Tread nmsg[2] offset[8] count[4]
Rread data[]
```

Reads from the implied fid. This permits multiple replies to the request. The field *nmsg* specifies a limit on the number of replies, and *count* refers to each single reply. The *data* field does not include a length, because the length is implied by the length of the entire operation.

Reads for directories return, for each file, the size of the metadata (size[4]) followed by the encoded metadata. The format used to encode the metadata is a series of name and value pairs, one for each attribute. A read does not return partial metadata for a file.

```
Twrite offset[8] endoffset[8] data[]
Rwrite offset[8] count[4]
```

Replaces the data between the two provided offsets with the one included in the request. The reply includes the offset used for the actual write, which may be useful for append-only files.

```
Tclunk
Rclunk
Tremove
Rremove
```

Clunk forgets the implied fid. *Remove* removes the file and clunks the fid.

```
Tattr attr[s]
Rattr value[s]
```

Returns the value for the file attribute named by *attr*. Attribute values are strings, encoded in UTF-8. Values for integer-valued attributes are strings with the printed value in decimal. The name ''*'' may be used to query for all attribute names. Its value is the set of attribute names separated by a space character.

```
Twattr attr[s] value[s]
Rwattr
```

Modifies the attribute *attr* to have a new *value*.

```
Tcond op[1] attr[s] value[s]
Rcond
```

The *cond* request converts IX requests into a microlanguage capable of making decisions. Here, *cond* is a relational operator and the *stat* attribute provides a name and value for an attribute. The server is expected to apply the relational operator to such attribute, and reply with *Rcond* only if the condition holds. Otherwise, the server replies with an error indication: *false*.

```
Tmove dirfid[4] newname[s]
Rmove
```

Moves the file identified by the implied fid to the directory and name indicated here.

12. Early Evaluation

We describe performance experiments in the order they were made, along with any optimization made to Mark II as a result.

NB: The second experiment was performed after using the file system for a while from a single client, with a warm cache. Its purpose was just to spot huge bottlenecks in the implementation. It is included here only to document what we found, but it should be repeated with more relevant workloads (e.g., those used in all remaining experiments).

All but the first two experiments are the result of 50 measures, reporting the average and variance for system, user, and elapsed time in seconds. In all of them the file system was mounted through a TCP connection within the machine running the file server program (to avoid most of the network interference). The scripts found in *bench* can be used to easily reproduce them.

As a reference, the same experiment was performed using *fossil*, mounted in the same way:

- */n/alboran* refers always to the *fossil* root.
- */n/9pix/root* refers always to the *creepy* root.

12.1. Profiling

This is the result of *tprof* after executing

```
; while(;){ cd /n/9pix/root/fos ; mk -a }
```

as a loop to compile the source code of fossil in a directory kept in Creepy for a few minutes. Reading data and synchronizing are the hot spots, or seem to be.

```
total: 7140
TEXT 00001000
    ms      %   sym
  1680   23.5   _profin
  1560   21.8   _profout
   440    6.1   _tas
   430    6.0   _mainp
   370    5.1   readn
   200    2.8   _threadrendezvous
   150    2.1   _callpc

   140    1.9   lock
   130    1.8   runthread
   110    1.5   dequeue
    90    1.2   ainc
    80    1.1   canexec
    80    1.1   longjmp
    70    0.9   _sched
    60    0.8   qunlock
    60    0.8   adec
    60    0.8   mbput

    50    0.7   xmbget
    50    0.7   threadsetname
    50    0.7   alt
    50    0.7   memmove
    40    0.5   xmunlink
    40    0.5   qlock
    40    0.5   _schedinit
    40    0.5   rpcworker9p
    30    0.4   _ifmt
    30    0.4   unlock

    30    0.4   putpath
    30    0.4   _div64by32
    30    0.4   rendezvous
    30    0.4   _threadgetproc
    30    0.4   setjmp
    30    0.4   xqlock
    30    0.4   workproc
    20    0.2   memset
    20    0.2   pwrite
    20    0.2   fidread
    20    0.2   fmtfmt
    20    0.2   dofmt
    20    0.2   _fmtcpy
    20    0.2   strlen
    20    0.2   strcmp
    20    0.2   pooldel
    20    0.2   dfwalk
    20    0.2   poolalloc
    20    0.2   poolreallocl
    20    0.2   blockcheck

    20    0.2   _threaddebug
    20    0.2   enqueue
    20    0.2   ctlproc
    20    0.2   _threadready
    20    0.2   altexec
    20    0.2   poolfreel
    20    0.2   listdelete
    20    0.2   xrwlock
    10    0.1   pchanged
    10    0.1   anew

    10    0.1   xcanwlock
    10    0.1   pathfmt
    10    0.1   xwlock
    10    0.1   mbrenew
    10    0.1   dbget
    10    0.1   getmk
    10    0.1   cliworker9p
    10    0.1   putfid
    10    0.1   fidaccessok
    10    0.1   prenew

    10    0.1   fidopen
    10    0.1   dfpwrite
    10    0.1   mused
    10    0.1   poolrealloc
    10    0.1   B2D
    10    0.1   utflen
    10    0.1   dodiv
    10    0.1   ltreewalk
    10    0.1   getdsize
    10    0.1   incref
```

12.2. Locks

Using the profiling tools built into the program for measuring waits on locks we see the following after using the file system for a while, from a single client. We have added comments to describe the ones of interest.

The useful information here is that, with a warm cache, there is no need to wait on locks, other than waiting to write replies to the client (which is done one at a time). But note that equests were processed concurrently.

```
locks   pc  ntimes  ncant   wtime   mtime
src -n -s 0x18a87 9pixprof  # qlock 3667    0   0   0
src -n -s 0x82b7 9pixprof   # qlock 70815   0   0   0
src -n -s 0x11884 9pixprof  # qlock 1   0   0   0
src -n -s 0x11898 9pixprof  # qlock 1   0   0   0
src -n -s 0x1344d 9pixprof  # qlock 18212   6   161503100088679
#   xqlock(&cli->wlk); in 9p.c:666
src -n -s 0x4fc7 9pixprof   # qlock 675137  0   0   0
#   this is b->slk
src -n -s 0x8359 9pixprof   # qlock 660450  0   0   0
    mused

src -n -s 0x5bf2 9pixprof   # qlock 70815   0   0   0
src -n -s 0x173de 9pixprof  # qlock 3669    0   0   0
src -n -s 0x18f7e 9pixprof  # qlock 18212   0   0   0
src -n -s 0x408b 9pixprof   # qlock 70815   0   0   0
src -n -s 0x4c85 9pixprof   # qlock 612093  0   0   0
# this is the hash

src -n -s 0x17496 9pixprof  # qlock 22497   0   0   0
src -n -s 0x7c89 9pixprof   # rwlock    953805  0   0   0
#   file's rlock

src -n -s 0x188a3 9pixprof  # qlock 9   0   0   0
src -n -s 0x4d58 9pixprof   # qlock 70815   0   0   0
src -n -s 0x7caf 9pixprof   # rwlock    7340    0   0   0
src -n -s 0x17938 9pixprof  # qlock 3667    0   0   0
src -n -s 0x7d06 9pixprof   # qlock 167173  0   0   0
#   Alloc locks
src -n -s 0x4de1 9pixprof   # qlock 541278  0   0   0
#   block load lock (wait on MBin blocks)

src -n -s 0x13abe 9pixprof  # qlock 18212   0   0   0
src -n -s 0x4e64 9pixprof   # qlock 70815   0   0   0
src -n -s 0x7ddd 9pixprof   # qlock 25549   0   0   0
src -n -s 0x185f4 9pixprof  # qlock 7197    0   0   0
src -n -s 0x1277a 9pixprof  # qlock 1   0   0   0
src -n -s 0x4ec1 9pixprof   # qlock 70815   0   0   0
```

12.3. Plot Conventions

Unless said otherwise, the plots in the following sections follow the same conventions. They refer to 50 individual measurements. The average and the variance for those are reported for both Creepy and Fossil. Times are in seconds and correspond to the user, system, and real time for the experiment.

In the graphics, each measure (in seconds, of real time) is plotted as a bullet for Creepy and as a circle for Fossil. Note that each individual measurement is subject to noise in the experiment, but the entire plot gives more information.

Most of such plots usually show that times remain in the same band, and do not seem to increase due to degradation of the file system during time after repeating the experiment multiple times. We say usually because, thanks to these experiments, a bug was found and fixed (see section 12.6).

Along with the plot for individual measurements, the average is marked with a solid line for Creepy and a dashed line for Fossil.

A whisker diagram is also depicted. It shows a box for the Q1 and Q3 quartiles, with a mark for the median (Q2), and two whisker indicating the error (i.e., the amplitude of the noise).

12.4. Creating a Tree

The tree is *$home/bin*, with 5847 Kbytes. In fossil:

```
Time 'mkdir /n/alboran/t$i && dircp /usr/nemo/bin /n/alboran/t$i'
avg 0.021 0.2678 2.1224
var 0.0160675 0.0594976 0.183454
```

In creepy:

```
Time 'mkdir /n/9pix/root/t$i && dircp /usr/nemo/bin /n/9pix/root/t$i'
avg 0.0234 0.329 1.6486
var 0.0149298 0.0637998 0.180464
```

The figure shows individual measurements.

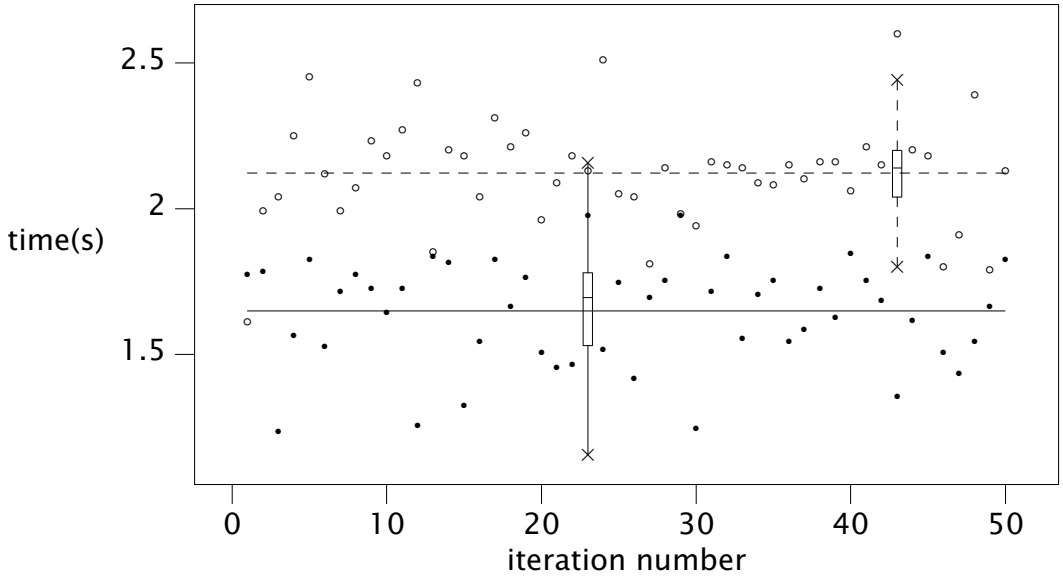

Figure 9 Creating a tree. Bullets and solid lines are for Creepy; Circles and dashed lines are for Fossil.

12.5. Reading a Tree

In fossil:

```
Time 'diff -nr /usr/nemo/bin /n/alboran/t$i'
avg 0.022 0.2214 4.4312
var 0.0139971 0.0498573 0.110816
```

In creepy:

```
Time 'diff -nr /usr/nemo/bin /n/9pix/root/t$i'
avg 0.0308 0.2468 2.4614
var 0.0167624 0.0464424 0.0508704
```

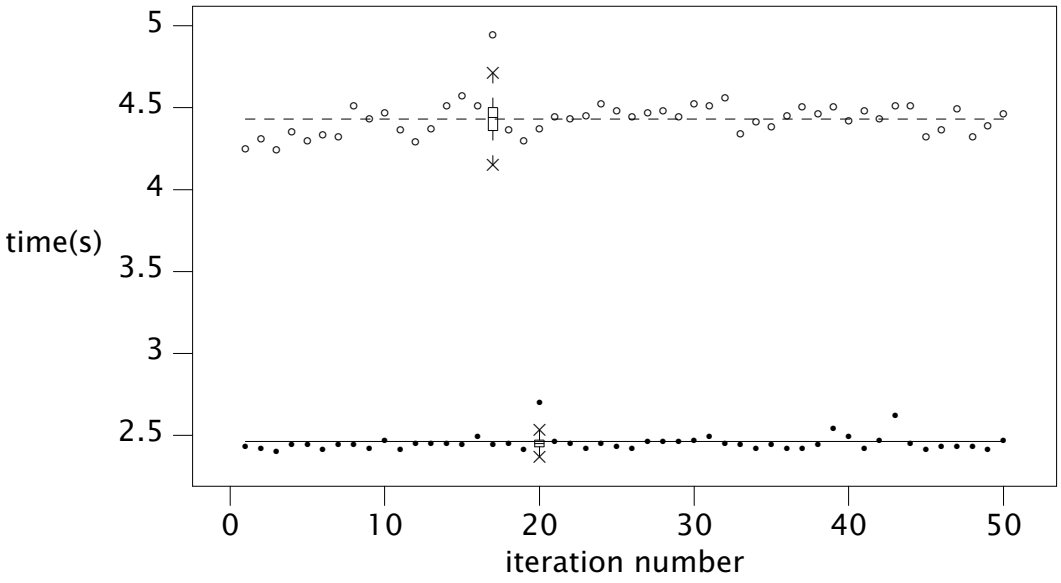

Figure 10 Reading a tree. Bullets and solid lines are for Creepy; Circles and dashed lines are for Fossil.

12.6. Removing a Tree

In fossil:

```
Time 'rm -rf /n/alboran/t$i'
avg 0.0018 0.0324 0.7378
var 0.00437526 0.0166059 0.128274
```

In creepy:

```
Time 'rm -rf /n/9pix/root/t$i'
avg 0.0016 0.0418 0.2164
var 0.00421852 0.0200703 0.00562792
```

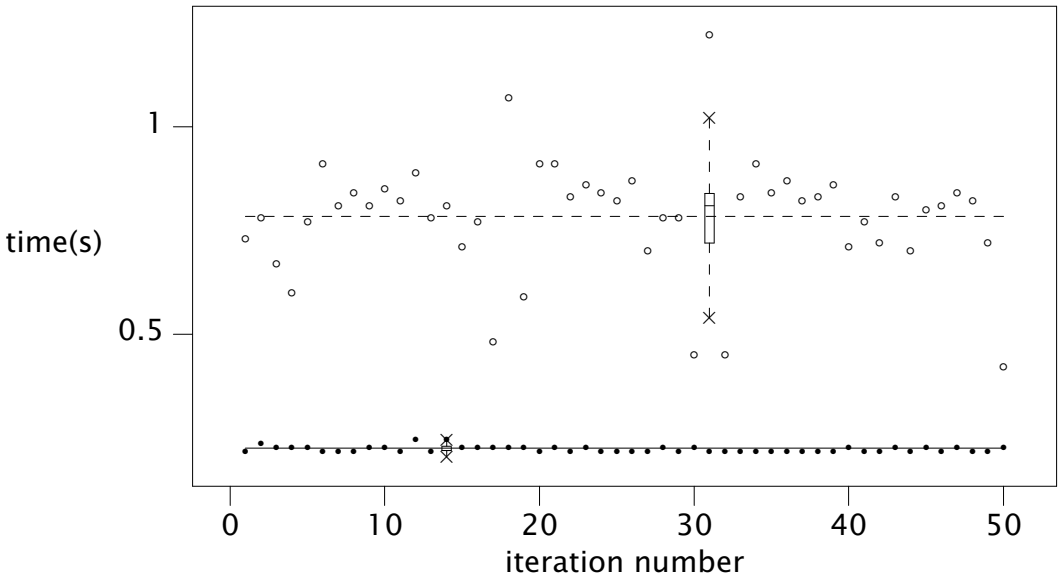

Figure 11 Removing a tree. Bullets and solid lines are for Creepy; Circles and dashed lines are for Fossil.

12.7. Copying a 128Mb file

In fossil:

```
Time 'fcp /tmp/bigfile /n/alboran/f$i'
avg 0.0334 2.4108 31.2906
var 0.019337 0.147743 1.04388
```

In creepy we obtained measures like:

```
Time 'fcp /tmp/bigfile /n/9pix/root/f$i'
avg 0.0435714 3.57714 13.9357
var 0.0312821 0.163398 0.560888
```

But then, after some time, it slows down to:

```
avg 0.037619 4.02524 30.5552
var 0.0144585 0.192266 1.2139
```

The cause was that multiple concurrent writes were being asked (for the entire tree) without waiting for the previous to complete. The implementation for the write is concurrent, one process assigns addresses and a different one writes to disk. This placed some pressure on the disk. Forcing a single file system write at a time fixed this, as described later.

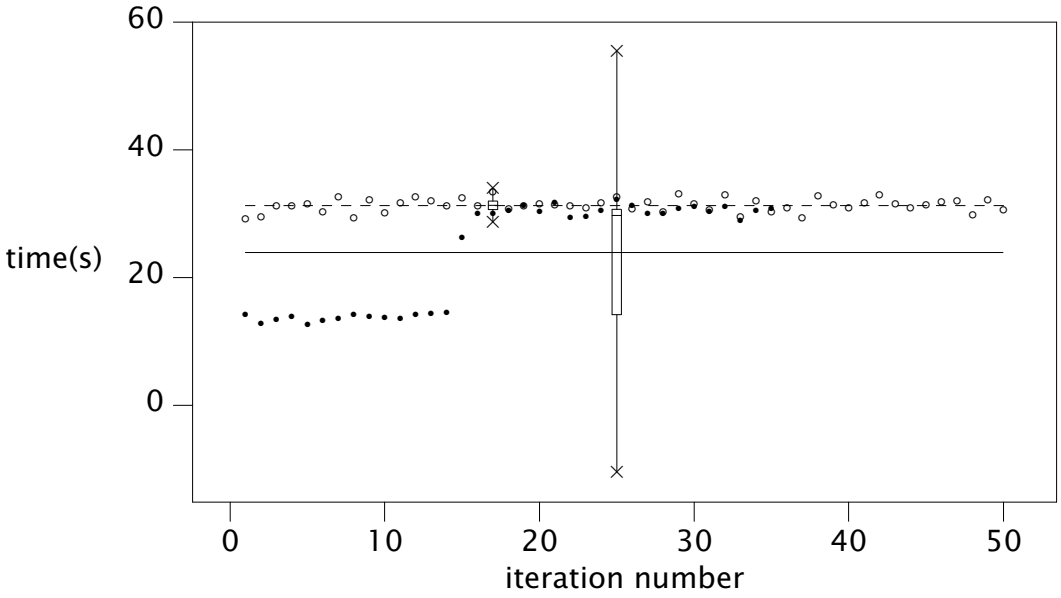

Figure 12 Copying a big file (before fixing a bug in Creepy). Bullets and solid lines are for Creepy; Circles and dashed lines are for Fossil.

12.8. Optimizations

As a result of the previous experiment, we made the following fixes:

1 Adding synchronization to permit a single file system snapshot (write) at a time.

2 Using multiple block write processes instead of a single one.

12.9. Copying a 128Mb file. Repeated.

This is the result copying to and from the file system (not relying on an external file system for the source), and using a fixed *creepy* as just described.

Using fossil:

```
Time 'fcp /n/alboran/f0 /n/alboran/fx'
avg 0.0748 6.9888 63.6284
var 0.0234734 0.211213 0.463013
```

Using creepy:

```
Time 'fcp /n/9pix/root/f0 /n/9pix/root/fx'
avg 0.0344 3.9614 9.8184
var 0.0157998 0.255079 0.819563
```

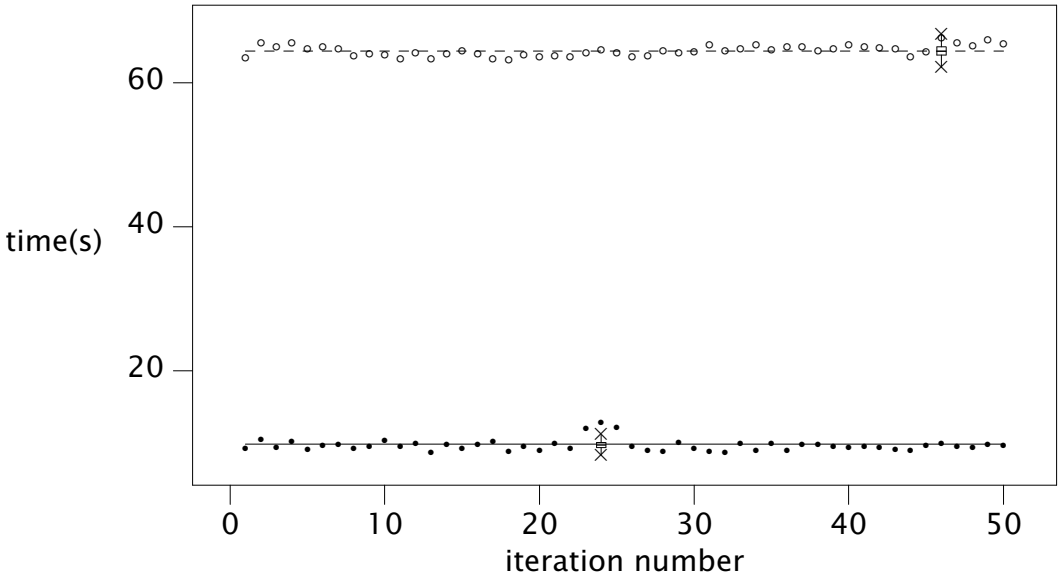

Figure 13 Copying a big file (after fixing a bug in Creepy). Bullets and solid lines are for Creepy; Circles and dashed lines are for Fossil.

12.10. Failed Optimization

We modified the file system to use a different process to release unused blocks (after a truncate or remove). The result did not differ and thus the modification was undone.

12.11. Concurrent usage

Compiling concurrently all source files of *fossil*(4) ussing fossil:

```
Time 'cd /n/alboran/fos && NPROC=8 mk -a>/dev/null'
avg 0.5664 0.7896 1.2502
var 0.0704725 0.084948 0.103677
```

Using creepy:

```
Time 'cd /n/9pix/root/fos && NPROC=8 mk -a>/dev/null'
avg 0.5248 0.7956 1.0652
var 0.0498258 0.0587336 0.036714
```

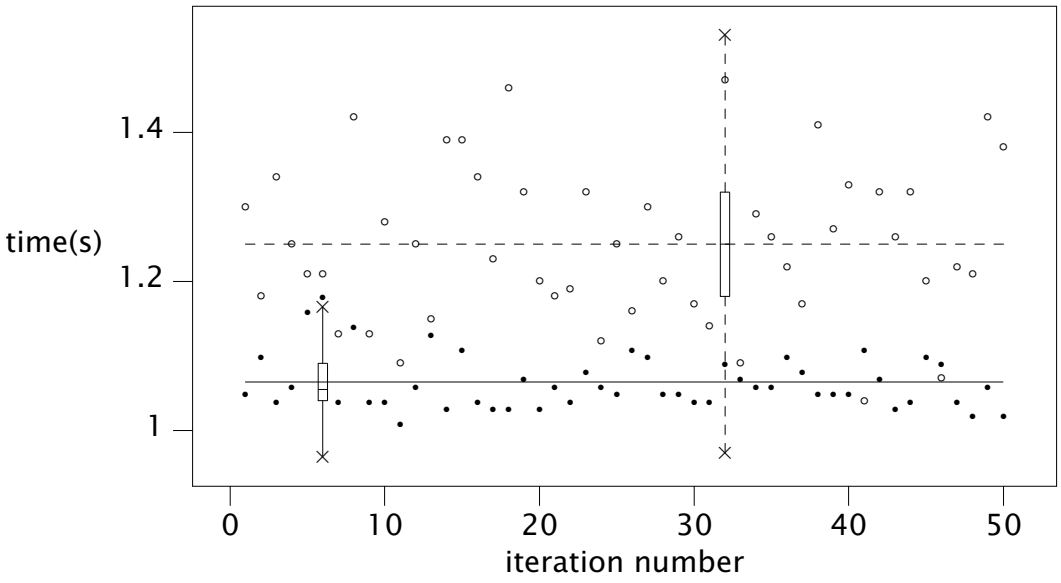

Figure 14 Compiling fossil source code with NPROC=8. Bullets and solid lines are for Creepy; Circles and dashed lines are for Fossil.