LA-UR-11-11422
Approved for public release; distribution is unlimited.

Title:	Gostor: Storage beyond POSIX
Author(s):	Ionkov, Latchesar A.
Intended for:	6th International Workshop on Plan9, 2011-10-20/2011-10-21 (Madrid, , Spain)

Gostor: Storage beyond POSIX

Latchesar Ionkov
*Los Alamos National Laboratory**
lionkov@lanl.gov

ABSTRACT

Gostor is an experimental platform for testing new file storage ideas for post POSIX usage. Gostor provides greater flexibility for manipulating the data within the file, including inserting and deleting data anywhere in the file, creating and removing holes in the data, etc. Each modification of the data creates a new file. Gostor doesn't implement any ways of oganizing the files in hierarchical structures, or mapping them to strings. Thus Gostor can be used to implement standard file systems as well as experimenting with new ways of storing and accessing users' data.

1. Introduction

Currently there are two popular ways of storing data – hierarchical file systems, and relational databases. The hierarchical filesystems consist of **files** (arrays of bytes) that have names. The files are grouped in **directories** which can be members of other directories. In order to access a file, one needs to know the **path**, i.e. the list of directories from **root** of the filesystem. Files data can be overwritten, but new data can be added only at the end of the file. Most of the popular file systems support the POSIX (or similar) set of file operations [2]. Relational databases enforce strong structure on the stored data, by enforcing it to be fitted in a set of tables. The data can be accessed by using the SQL language. Relational databases provide better consistency and concurrency guarantees. They are hard to install and maintain, often requiring professional database administrator in order to achieve reasonable performance.

There are many attempts to introduce some of the database benefits to the file systems, such as transactions, snapshots, etc. Even though some of the popular filesystems support snapshots and versioning, creating and manipulating snapshots is not part of the standard file operations.

In many cases, especially with scientific datasets, extending files only at the end is too restrictive. To avoid the limits, some scientific data formats, like HDF [1] reimplement most of the file system data structures in the file.

There are some attempts to go beyond both hierachical namings of files as well as relaxation on the restrictions on how the content of the files can be modified. hFAD [4] allows data to be inserted and deleted at any place in the file. It also removes the dependency on file names when accessing the files and allows adding tags that describe the content of the files.

Most of the attempts to change the way data is stored and accessed are built on top of standard file systems or databases.

Gostor is a low-level storage system that provides consistent way of manipulating arrays of bytes, providing both versioning support as well as ability to insert, delete and modify data anywhere within the array. It also allows insertion and removal of holes in the array.

2. Gostor Architecture

Gostor provides interface for

*LANL publication: LA-UR-XX-XXXX

2.1. Files

A **file** in Gostor is an immutable string of bytes. Regions of file where data haven't been written are holes in the file. They are not stored and don't use disk space. Reading from a hole region returns zeroes.

A file is identified with a 64-bit integer called **fileID**. The fileID may change during the life span of the file. If the fileID is changed, Gostor guarantees that all references to the old fileID within the storage will be modified, and that the old fileID can be used until they are specifically discarded, or the connection to Gostor is closed. There is no guarantee that a fileID retrieved during previous connection will still be valid when a new connection is established.

File with fileID 0 always points to a zero-length file.

2.2. Data Content Types

When reorganizing the disk space, Gostor needs to know if any of the files contains references to other files. For that reason all data stored is assigned **data content type**. When writing data to files, the user can specify whether the data contains references to other files. Because data type is specified by write, there is no restriction for the whole file to have the same data content type. Gostor can coalesce content of data written by multiple writes as long as it is sequential and of the same type. When data is read, Gostor returns only data of the same content type.

Currently there are only two data types available to the user:

Data Type	Description
BTData	Data that doesn't contain any fileIDs
BTDir	File content with only fileIDs

Gostor defines additional data content types that are used internally to implement the file layout.

Gostor allows up to 32 thousand data types defined, but currently doesn't provide operations in the protocol for creating new data types. Extensions to Gostor (such as filesystems) can use an internal API.

2.3. Operations

2.3.1. Read

```
read(file, offset uint64, buf []byte) (count int, dtype uint16,
    err os.Error)
```

The **read** function reads up to length of the `buf` array bytes from the specified file, starting from the specified `offset` and stores it in the `buf` array. It returns the number of bytes returned as well as the data content type of the data. If an error occures while reading, `err` contains the error.

Read can return less than `len(buf)` bytes in two cases – if there is no more data in the file (i.e. `offset + len(buf)` is greater than the file's size), or if the data in the file has mixed data content types.

2.3.2. Write

```
write(file, soffset, eoffset uint64, count uint32, data []byte,
    dtype uint16) (newfile uint64, err os.Error)
```

Because Gostor files are immutable, each write operation creates a new file. It receives an existing file as argument, clones its content, applies the changes of the data content and returns the fileID of the newly created file.

The write call is used to do any modifications to a file. It replaces the data currently located between offsets (`soffset, eoffset`) with `count` bytes containing the data from the `data`

array. If the length of the `data` array is less than `count`, a hole is created at the end of the region. Table 1 shows examples on how the Write operation can be used to modify the data.

Operation	Description
`write(.., 10, 10, 100, data)`	Insert 100 bytes of data at offset 10
`write(.., 10, 10, 100, nil)`	Insert 100 byte hole at offset 10
`write(.., 10, 110, 0, nil)`	Delete 100 bytes of data starting at offset 10
`write(.., 10, 110, 100, data)`	Overwrite 100 bytes of data starting at offset 10

Table 1: Examples on data modifications using Write

2.3.3. Size

```
size(file uint64) (size uint64, err os.Error)
```

Returns the size of the specified file.

2.3.4. Forget

```
forget(file uint64)
```

Informs Gostor that the user is no longer going to use the file with the specified fileID. If there are no more references to the file, it may be garbage-collected.

3. Current Implementation

Gostor is implemented in Go. Currently it is approximately 2000 lines of code.

3.1. Segments and Blocks

Gostor uses log-structured data layout, similar to the ones used in log-structures filesystems [3]. The disk space is divided into segments, and the segments are kept in a doubly-linked list. The segments can be **free**, **full**, or **active**. There is only one active segment at a time. All new data is appended at the end of the active segment. Once the segment is full, it is marked as "full" and the next segment in the list is set as active.

The space within the segments is not divided into fixed-size chunks. Each segment consists of a number of variable-sized blocks. Each block can be up to 2^16 bytes long (including the header). The header of the block contains its size as well as the type of the data it contains (data content type). The blocks always start at an even offset.

Figure 1 shows the physical layout of the segments and the blocks.

3.2. Files

Gostor uses modification of Btrees to describe files. Blocks at level 0 contain the data of the file. Blocks at level 1 contain entries to the data blocks at level 0, and so forth. Because Gostor allows insertion of data anywhere in a file, offsets to data are not constant and can't be stored in the intermediate Btree blocks. Instead, each Btree entry contains the size of the data it describes. Normally the Btree blocks are fixed size and waste some space when there are not enought entries. Because of the log-structured layout used by Gostor, the Btree representation we use is compact and doesn't waste any disk space.

The fileID ofa file is the offset (relative to the beginning of the disk) of the block describing the root of the file Btree. This approach simplifies the implementation of Gostor and avoids a level of indirection, that is common to inode based systems.

Figure 2 shows an example of a file with fileID 348. The file's Btree has tree levels. The first entry in the root block points to block 388 and defines that that block describes the first 1100

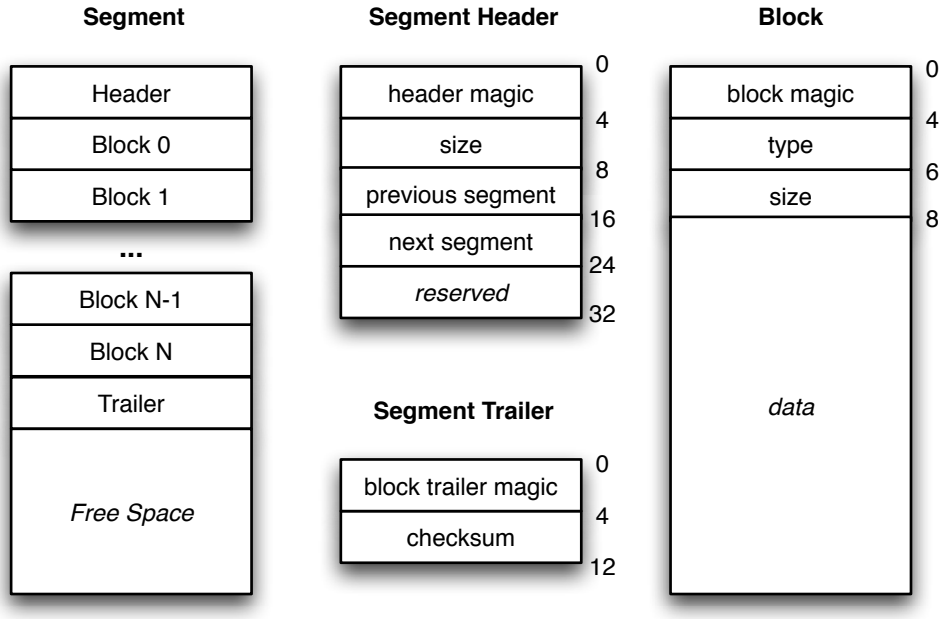

Figure 1: Gostor Disk Layout

bytes of the file. Block 388 has two entries, one is 700 bytes long and the data is stored in block 500. The second entry describes 400-byte long hole (the block is 0). Figure 2 also shows how the file blocks are laid out in a segment.

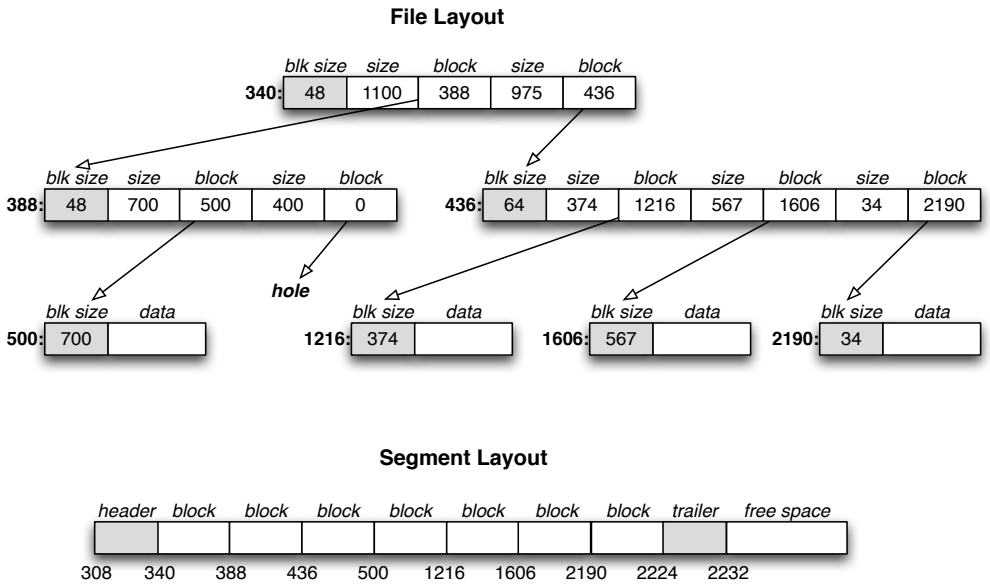

Figure 2: Btree and Segment Representation of a File

3.3. Operations

The Write operation uses the specified file's Btree as a base for the Btree of the newly created file. It creates new blocks for a subtree of the file, up to the new file's root block. While building a new tree, Gostor tries to coalesce neighboring blocks if their size is less than the allowed size (2^16 bytes for level 0 blocks and 8192 bytes for intermediate blocks). Gostor fills up the intermediate blocks with entries left-to-right so the rightmost affected block on the level is left unfilled. This approach is optimized for the most-frequently use case when data is appended at the end of the file.

Figure 3 shows the layout of the data when operation `write(340, 700, 700, 162, data)` is performed on the file shown on Figure 2. The operation inserts 150 bytes of data at offset 700 and increases the size of the hole with 12 bytes.

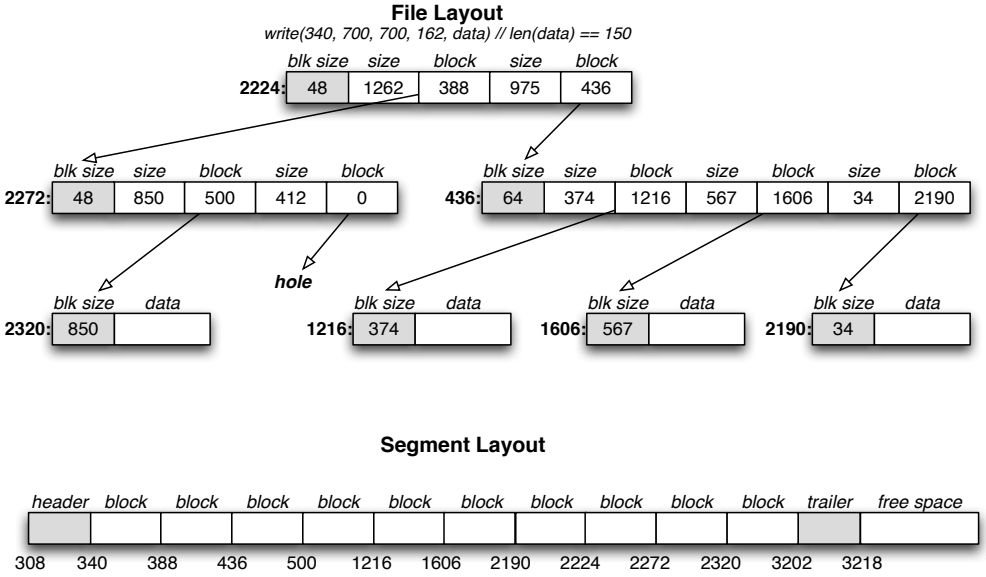

Figure 3: Btree and Segment Representation of a file after Write operation

3.4. Garbage Collection

Each Write operation in Gostor creates a new file with a new root block, some intermediate Btree blocks as well as some data blocks. In most cases, once the new file is created, the old file is no longer useful and the blocks that belong only to it should be recycled. Gostor has knowledge of all references to blocks stored in it. It also keeps track of the blocks that the user is using at the moment. **Forget** operation should be called if the user no longer cares about a block.

Having information which blocks are still in use allows Gostor to reclaim the space that is no longer accessible. Currently Gostor supports only in-memory garbage collection. When a block is stored in Gostor, it is added to an in-memory list of data to be written to the disk. The data is kept in the memory for some time, giving it a chance to get obsoleted by subsequent changes to the list of live files.

When the user no longer going to use a reference to a block, the **forget** command should be used to upate Gostor's knowledge of what data is regarded as live by the user.

Currently Gostor supports only in-memory garbage collection. Once the data is committed to the disk, it can't be removed. Writing a garbage collector for the on-disk data is planned as future work.

3.5. Committing Blocks to Disk

When a block is stored in Gostor, it is not written to the disk right away. In most cases, the data in the block will be obsoleted soon by subsequent call and premature write to the disk will be wasteful. Gostor keeps the blocks in memory for some time to allow the data to be made obsolete. Gostore doesn't have knowledge what blocks will be actually stored on disk and therefore can't assign disk offsets for the blocks before they are actually committed. Instead, Gostor returns temporary offsets that can be used to access the data, or store references to the block in Gostor.

Before committing the data on disk, Gostor runs the garbage collector to free all blocks that are no longer live. Then it assigns disk offsets to the remaining blocks, and updates the temporary offsets stored in the blocks to the permanent ones. It can do that because it has knowledge if a block contains references to other blocks. Once the references are updated, the blocks are stored on the disk. The temporary offsets can be used until they are "forgotten".

4. Future Work

The current Gostor implementation doesn't have garbage collector that can reclaim data already written to the disk. There are plans for implementing copying, generational garbage collector that crawls through the existing data and copies the reachable blocks into new segments. Because of the high penalty of reading all existing data, the future implementation might be modified so in segregates blocks that contain references to other blocks in separate segments.

Once the Gostor prototype is stable enough, we plan to implement conventional hierarchical filesystem on top of it, as well as experiment with porting existing formats (HDF5) for scientific data on top of it.

References

[1] HDF Hierarchical Data Format. http://www.hdfgroup.org.

[2] *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). 1990.

[3] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 25:1–15, September 1991.

[4] Margo Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.