ECE 5984 Virtualization Technologies
Project 1: Unikernel Filesystem

Team1
A K M Fazla Mehrab
Mincheol Sung

Introduction

HermitCore is a small and simple unikernel which depends on host operating system for system calls. When a system call is issued the call is forwarded to host operating system which executes the system call and returns the results. As a result, it introduces a large syscall forwarding overhead.

HermitCore doesn't have any own filesystem. So for any filesystem operation it depends on host operating systems filesystem. As filesystem operations are dependent on number of syscalls, this syscall forwarding works as bottleneck for faster file manipulation. The goal of this project is to come up with a simple filesystem for HermitCore so that the need of forwarding the filesystem related syscalls can be avoided.

For this project we used RAM as target disk. We implemented a file system with directory support and scalability. We implemented all the required semantics of the filesystems so that our guest operating system does not require any host involvement for filesystem operations. We ran multiple benchmarks to evaluate the performance of the implemented filesystem along with other standard filesystem.

In the rest of the report we are going to describe implemented filesystem design, detail on implementation, description about how to build modified HermitCore and newlib and run the benchmarks, experimental setup and results and our analysis as well as interpretation of the results.

Detailed description of filesystem design

For any filesystem we can divide the data into to categories: metadata and file data. Metadata is called the data of the data i.e. information of the file where file data is the data corresponds to file. We implemented the whole filesystem on RAM. So for metadata we maintained an in-memory metadata by storing the in structures. On the other hand, for file data we allocated chunk of memory to be used as disk.

Metadata

1. Disk (struct disk): struct disk is a structure that represents physical disk (but in the main memory). It has four components.
    a. Root directory: a directory descriptor struct describing the root directory
    b. FAT: File Allocation Table, explained below
    c. Data block: a list of data blocks which contain data
    d. Free block list: a list of free blocks which not contain data and available for writing data

2. File Allocation Table (FAT): FAT defines relation between a file and data blocks in the disk (right table of **Figure 1**). FAT entry has index of FAT, data block index indicating data block in the disk, and next index indicating next FAT entry. A file can be read or written more than one block by traveling the FAT.

3. File descriptor structure (struct file_desc): file descriptor table is a metadata of a single file or directory. It has file's name, offset, size, mode, flags, status, and index of first FAT entry.

4. Directory descriptor structure (struct dir_desc): a metadata of a directory. It is a table containing file descriptor table of files or directories in the directory (left table of **Figure 1**).

5. File descriptor table (struct file_desc **): a file descriptor table. Its entry has points to a file descriptor of file that the file descriptor structure can be reached by an index of file descriptor. File descriptor table's index can be treated as file descriptor id (as known as int fd).

6. Free file descriptor id list (struct free_file_desc_id): a list of file descriptor IDs (as known as int fd) which are available to be assigned to a file. When a file is opened, it gets a file descriptor id from the list.

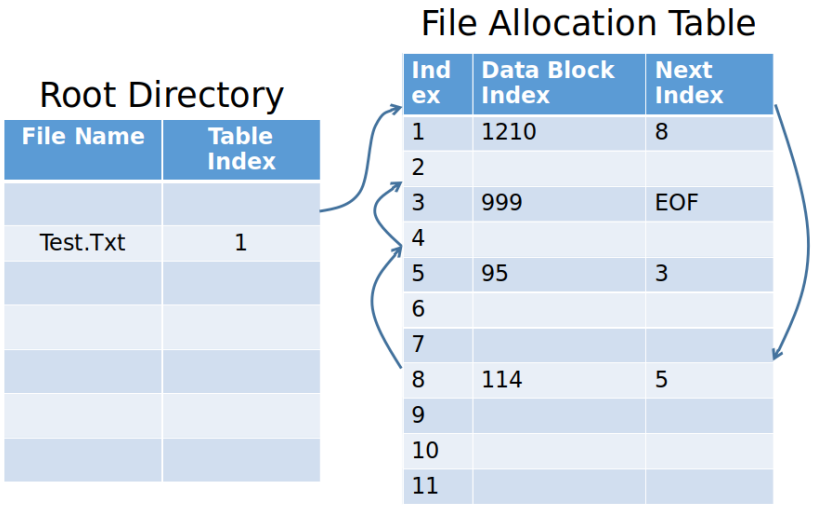

Figure 1: Root directory descriptor table and File Allocation Table

System calls supports

int sys_open (const char *pathname, int flags, mode_t)

1. Description: Open a file
2. Arguments
    a. pathname: opens the file specified by pathname
    b. flags: The flags can specify accessing mode such as O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC and O_APPEND
        i. O_RDONLY: Read only
        ii. O_WRONLY: Write only
        iii. O_RDWR: Read and only
        iv. O_CREAT: Create the file if not exists
        v. O_TRUNC: If the file already exists and is a regular then allow writing
        vi. O_APPEND: Open file with setting its offset to the end of file
    c. mode_t: permission set when created (e.q., 777, 455)
3. Return: returns file descriptor id
4. Behavior: first check flags whether creation or not. If flags is O_CREAT, allocate metadata of a file and set to initial values. In addition, obtain free file descriptor and store it in the file

descriptor table. If not O_CREAT, for example O_RDWR, it searches file descriptor of the file from the file descriptor table and return the file descriptor.

int sys_close (int fd)

1. Description: close a file
2. Arguments
    a. fd: file descriptor of file to be closed
3. Return: zero on success, otherwise returns non zero integer
4. Behavior: deletes file descriptor from file descriptor table

int sys_mkdir (const char* path, unsigned int mode)

1. Description: create a directory
2. Arguments
    a. path: create a directory specified by path
    b. mode: permission set when created
3. Returns: zero on success
4. Behavior: create directory with directory table and initialize metadata

int sys_rmdir (const char* path)

1. Description: delete a directory
2. Arguments
    a. path: delete a directory specified by path
3. Returns: zero on success
4. Behavior: deletes directory and release directory table, and metadata

int sys_read (int fd, void* buffer, size_t len)

1. Descriptor: reads data from a file specified by fd and stores to buffer
2. Argument
    a. fd: file descriptor of the file to be read
    b. buffer: buffer which store data on
    c. len: size of data to be read
3. Returns: size of data read
4. Behavior: copies date in the data block in to the buffer. Particularly, it accesses to the last target data block and traverses to the front.

int sys_write (int fd, void* buffer, size_t len)

1. Description: write data on a file specified by file descriptor
2. Arguments
    a. fd: file descriptor of the file to be written on
    b. buffer: buffer contains data to be written
    c. len: size of data to be written
3. Returns: size of data written
4. Behavior: like read, write date to the last target data block and then to the front. If size of data is bigger than a block, it gets free blocks from the free block list.

off_t sys_lseek (int fd, off_t offset, int whence)

1. Description: set file's offset
2. Arguments
    a. fd: file descriptor of the file which the offset is set on
    b. offset: offset to be set
    c. whence
        i. SEEK_SET: The file offset is set to offset bytes
        ii. SEEK_CUR: The file offset is set to its current location plus offset bytes
        iii. SEEK_END: The file offset is set to the size of the file plus offset bytes
3. Returns: offset set by lseek
4. Behavior: set input to offset of file descriptor structure

int sys_unlink (const char *path)

1. Description: delete a file
2. Arguments
    a. path: deletes the specified by path
3. Returns: zero on success
4. Behavior: finds the file descriptor of the file in the directory or path, releases data blocks and adds the data blocks to free block list.

char * sys_getcwd(char *buf, int max_line)

1. Description: get current working directory
2. Arguments
    a. buf: store results on the buf
    b. max_line: dummy for API compatibility
3. Returns: buffer containing the directory path
4. Behavior: our filesystem always works on root directory, so simply return "/"

char * sys_ls (const char *path)

1. Description: list files in a directory
2. Arguments
    a. path: the directory specified by path
3. Returns: null
4. Behavior: this functions is for debugging purpose only. Nothing is returned. Only prints log messages on the HermitCore.

Filesystem implementation

We add extra system calls (mkdir, rmdir, getcwd) to newlib. For others (open, close, mkdir, rmdir, read, write, lseek, and unlink), we can simply modify newlib's implementation. In newlib, functions are calling sys_* system calls of HermitCore. System calls are implemented on HermitCore/kernel/syscall.c and declared on HermitCore/include/hermit/syscall.h
In system calls implementation, each system calls are calling our filesystem functions such as hfs_read, hfs_write on HermitCore/fs. In addition, our filesystem functions and structures are declared on HermitCore/hfs.h and HermitCore/hfs_struct.h.

In main.c, initd functions allocate ramdisk (struct disk) then calls hfs_mkfs which creating and initiating our file systems. Function hfs_mkfs on HermitCore/fs/hfs.c is a function that creates metadata and fills them initial values. After hfs_mkfs is done, our filesystem is set to work.
We split our functions into each file (e.q., HermitCore/fs/hfs_open.c, HermitCore/fs/hfs_lseek.c). Furthermore, some extra functions such as pop_free_block (get a free block from the free block list) are implemented on HermitCore/fs/misc.c and only called by our filesystem functions (hfs_* prefix functions). Thanks to unikernel's single memory space, the filesystem metadata can be declared as global and easily accessed by our filesystem functions without any special operation.

Build and Run

For building HermitCore and newlib we assume that all the corresponding prerequisite packages, including HermitCore Toolchain, Cmake, and so one are installed are described in provided Helper Guide for this project. Then set the environment variable if not done so far as follows.

```
$ export PATH=$PATH:/opt/hermit/bin
```

Our submission comes with two patches: one for HermitCore modifications and another for newlib modifications.
Download HermitCore, apply patches, compile and install as shown below.

```
$ git clone https://github.com/RWTH-OS/HermitCore.git hermit
$ cd hermit
$ git submodule init
$ git submodule update
$ git apply HermitCore.patch
$ mkdir build
$ cd build
$ cmake ..
$ make -j `nproc` && sudo make install
```

(Comment: there are trailing white line warning during applying the patch. We couldn't fix the warning due to time limitation.)

Download newlib, apply patches, compile and install as shown below.

```
$ git clone https://github.com/olivierpierre/newlib.git
$ cd newlib
$ git apply newlib.patch
$ ./configure --target=x86_64-hermit --prefix=/opt/hermit --disable-
shared --disable-multilib --enable-lto --enable-newlib-hw-fp --
enable-newlib-io-c99-formats --enable-newlib-multithreaded
$ make -j `nproc`
$ sudo PATH=$PATH:/opt/hermit/bin make install
```

For this project we ran two benchmark, postmark and latency_test, which is describe in Experimental Description section in detail. Sources for these benchmarks should be available in hermit/usr/tests directory. You can compile and run with make.

```
$ cd HermitCore/usr/tests/
$ make postmark
```

Then it prompts you to set configuration for postmark. If you want to run with default configuration, then just type "run".

```
PostMark v1.5 : 3/27/01
pm> run
```

Similarly we can build and run our micro benchmark for latency measurement called latency_test.

```
$ make latency_test
```

It prompts you to set number of iteration, for example iteration 10,

```
Latency Test for
10
```

Experimental Description

A filesystem have different aspects to be evaluated. For our filesystem we have chosen two different criteria to evaluate its performance. First one is doing different types of transactions, like create file, delete file, read file, and append file, on the implemented file system and get the elapsed times. For this purpose we used an well-known benchmark called Postmark[1,2]. It gives us evaluation results with following information.

- Elapsed time
- Elapsed time spent performing transactions and average transaction rate (files/second)
- Total number of files created and average creation rate (files/second)
- Number of files created initially and average creation rate (files/second)
- Number of files created during sequence of transactions and average creation rate (files/second)
- Total number of files read and average rate (files/second)
- Total number of files appended and average rate (files/second)
- Total number of files deleted and average deletion rate (files/second)
- Number of files deleted after transactions were complete and average deletion rate (files/second)
- Number of files deleted during sequence of transactions and average deletion rate (files/second)
- Total size of data read and average input rate (bytes/second)
- Total size of data written and average output rate (bytes/second)

Second one is measuring the latency of the filesystem APIs. Here, as shown in the following **Figure 2**, by latency we mean time difference between the time of a filesystem API function call issues and the time when the function actually started execution. To do this experiment we selected a very lightweight function, called lseek(), which do minimum amount of operation and thus takes negligible amount of time to be executed, so that we can consider the total time of a lseek() operation as the approximate latency of the filesystem.

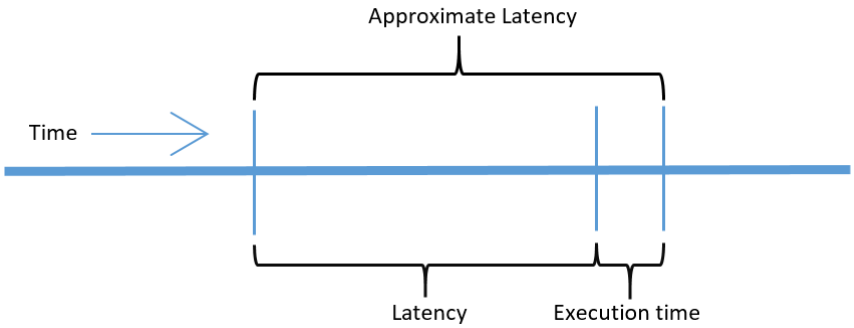

Figure 2: Approximate latency calculation for lseek operation

Experimental Setup:

Hypervisor: KVM
Guest OS: Ubuntu 16.04
CPU: AMD Opteron 63xx class
VCPU: 4
Memory: 8gb

Experimental Results:

1. Postmark

Postmark benchmark results for different configurations are presented below.

a. Default configuration

i. file size 500~10000
ii. Simultaneous: 500
iii. Transaction: 500
iv. Subdirectory: 0
v. Read_block_size: 512
vi. Write_block_size: 512
vii. Buffered i/o

	Filesystem for HermitCore (Our implementation)	Linux Filesystem (ext4)
Total Execution Time(s)	4	1
Transaction Time(s)	2 (250 per second)	1 (500 per second)
File Created	764 (191 per second)	764 (764 per sec)
File Created alone	500 (250 per second)	500 (500 per sec)
File Created with transaction	264 (132 per second)	264 (264 per sec)
File Read	243 (121 per second)	243 (243 per second)
File Append	257 (128 per second)	257 (257 per second)
File Deleted	764 (191 per second)	764 (764 per second)
File Deleted alone	528 (528 per second)	528 (528 per second)
File Deleted with transaction	236 (118 per second)	236 (236 per second)
Data Read(MB)	1.36 (349.31 kb per second)	1.36 (1.35 mb per second)
Data Write(MB)	4.45 (1.11 mb per second)	4.45 (4.45 mb per second)

Table 1: Postmark result with default configuration

b. Default configuration with 1000 files, not buffered_io

	Filesystem for HermitCore (Our implementation)	Linux Filesystem (ext4)
Total Execution Time(s)	7	1
Transaction Time(s)	2 (250 per second)	1 (500 per second)
File Created	1248 (178 per second)	1248 (1248 per second)
File Created alone	1000 (200 per second)	1000 (1000 per second)
File Created with transaction	248 (124 per second)	248 (248 per second)
File Read	255 (127 per second)	255 (255 per second)
File Append	245 (122 per second)	245 (245 per second)
File Deleted	1248 (178 per second)	1248 (1248 per second)
File Deleted alone	996 (996 per second)	996 (996 per second)
File Deleted with transaction	252 (126 per second)	252 (252 per second)
Data Read(MB)	1.28 (186.67 kb per second)	1.28 (1.24 mb per second)
Data Write(MB)	6.89 (1008.23 kb per second)	6.89 (6.89 mb per second)

Table 2: Postmark result with 1000 files and no buffered_io

c. Default configuration with 10,000 files, not buffered_io

	Filesystem for HermitCore (Our implementation)	Linux Filesystem (ext4)
Total Execution Time(s)	58	1
Transaction Time(s)	2 (250 per second)	1 (500 per second)
File Created	10256 (176 per second)	10256 (10256 per second)
File Created alone	10000 (181 per second)	10000 (10000 per second)
File Created with transaction	256 (128 per second)	256 (256 per second)
File Read	247 (123 per second)	247 (247 per second)
File Append	253 (126 per second)	253 (253 per second)
File Deleted	10256 (176 per second)	10256 (10256 per second)

File Deleted alone	10012 (10012 per second)	10012 (10012 per second)
File Deleted with transaction	244 (122 per second)	244 (244 per second)
Data Read(MB)	1.25 (22.13 kb per second)	1.25 (1.25 mb per second)
Data Write(MB)	51.36 (906.70 kb per second)	51.36 (51.36 mb per second)

Table 3: Postmark result with 10,000 files and no buffered_io

d. Default configuration with 100,000 files, not buffered_io

	Filesystem for HermitCore (Our implementation)	Linux Filesystem (ext4)
Total Execution Time(s)	1004	5
Transaction Time(s)	7 (71 per second)	1 (500 per second)
File Created	100238 (99 per second)	100238 (20047 per second)
File Created alone	100000 (112 per second)	100000 (25000 per second)
File Created with transaction	238 (34 per second)	238 (238 per second)
File Read	246 (35 per second)	246 (246 per second)
File Append	254 (36 per second)	254 (254 per second)
File Deleted	100238 (99 per second)	100238 (20047 per second)
File Deleted alone	99976 (908 per second)	99976 (99976 per second)
File Deleted with transaction	262 (37 per second)	262 (262 per second)
Data Read(MB)	1.22 (1.25 kb per second)	1.22 (250.21 kb per second)
Data Write(MB)	502.19 (512.19 kb per second)	502.19 (100.44 mb per second)

Table 4: Postmark result with 100,000 files and no buffered_io

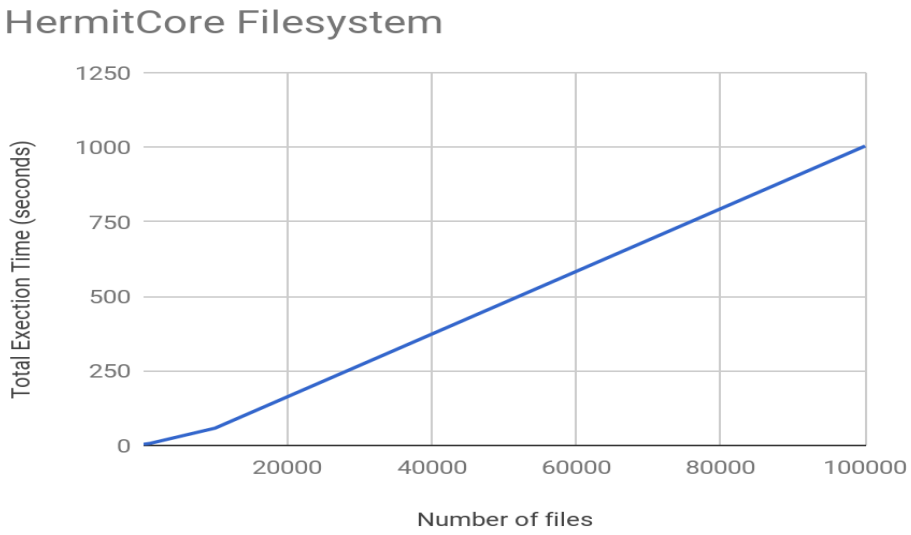

Figure 3: Total execution time vs Number of files for Postmark Benchmark

2. Latency of lseek() in **micro seconds**

Latency measurement micro benchmark results for different iterations are presented below.

iteration	Original HermitCore (Forward to linux)	Filesystem for HermitCore (Our implementation)	Linux filesystem (ext4)
1	1007	194	4
10	2652	313	9
100	17203	394	62
1000	130933	1031	599
10000	1199553	4736	2195
100000	11959842	30428	27503

Table 5: latency benchmark result of lseek() with several iterations

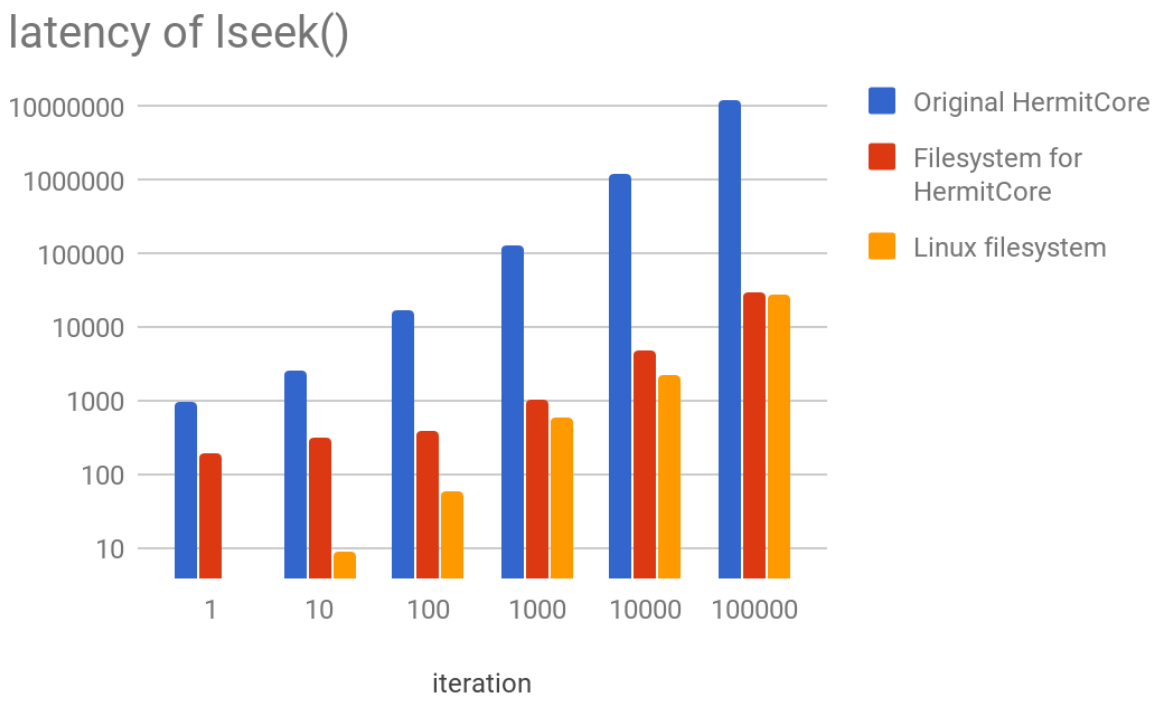

Figure 4: Approximate latency for different iteration on different OSs.

Experimental Result Analysis & Interpretation:

Postmark:

Postmark benchmarking shows that our implemented filesystem for HermitCore supports higher number of file creation and other operations including read, write, append and delete etc. In that sense we can say that our implemented filesystem is scalable. On the other hand, from the above data in tables (**Table 1~4**) and graph in **Figure 3** we can see that, with the increase of number of files, total execution time increases linearly which is also true for Linux Ext4 filesystem. But for implementation the curve is much stiffer than the Linux one.

One key factor for highly stiffer curve for implementation is simple table based (array) metadata such as FAT, and directory table (struct file_desc *dir_table on HermitCore/include/hermit/hfs_struct.h). Whenever file or directory is created/deleted, it should scan the table sequentially and this sequential scan increases the total execution time. For better scalability, we need to introduce faster and efficient data structures (e.q., directory tree) for filesystem metadata.

Latency:

From the above **Table 5** and graph in **Figure 4** we can see that with the increase of iterations the for each of the host and guest operating systems filesystem, HermitCore, Linux and our implementation on HermitCore, the latency increases linearly. The original HermitCore filesystem implementation requires most time to forward a filesystem syscall operation where native Linux requires the least. Time requirement of syscall forwarding, for our implementation of filesystem on ramdisk, lies in between.

From this scenario we can interpret that, original HermitCore filesystem, which is dependent on host system for forwarding to syscall, have the most latency which is 250 times (for one operation) to 435 times (for 100000 operations) higher compared to native Linux latency. On the other hand, our implementation, for HermitCore on ramdisk, is 30 times higher compared to native Linux latency for one syscall operation. But interestingly for 100000 iterations both latencies for our implementation and native Linux are almost same. So we can say that, for higher amount of filesystem transactions our implementation performs better than fewer transactions.

Conclusion

Our filesystem achieves the goal of this project which is to come up with a simple filesystem for HermitCore so that the need of forwarding the filesystem related syscalls can be avoided.
We implemented a file system with directory support and scalability that our filesystem can handle enormous number of files (~100,000). We implemented all the required semantics of the filesystems so that our guest operating system does not require any host involvement for filesystem operations. In terms of latency of a system call (lseek) our filesystem is slower than the Linux filesystem (e.q., ext4), but still much faster than current HermitCore's syscall forwarding scheme. In contrast, postmark benchmark results show that execution time increases as the number of files increases in our filesystem. We are sure that better performance can be expected with using efficient data structure for metadata instead of current table based metadata.

References

[1] http://www.dartmouth.edu/~davidg/postmark_instructions.html
[2] https://koala.cs.pub.ro/redmine/attachments/.../Katcher97-postmark-netapp-tr3022.pdf