EE516 : Project 4

Gaurav Kalra

(Student ID: 2016 45 93)

gvkalra@kaist.ac.kr

Task 1: Data encryption on FUSE file system

```c
int bb_read(const char *path, char *buf, size_t size, off_t offset, stru
{
    int retstat = 0;

    log_msg("\nbb_read(path=\"%s\", buf=0x%08x, size=%d, offset=%lld, fi
        path, buf, size, offset, fi);
    // no need to get fpath on this one, since I work from fi->fh not th
    log_fi(fi);

    retstat = pread(fi->fh, buf, size, offset);
    if (retstat < 0)
        retstat = log_error("bb_read read");
    else if (retstat != 0) // decrypt if not EOF
        enc_decrypt_data((unsigned char *)buf, (size_t)retstat); // prea

    return retstat;
}
```

Figure 1: bb_read(), decrypt data after pread()

```c
int bb_write(const char *path, const char *buf, size_t size, off_t offset,
        struct fuse_file_info *fi)
{
    int retstat = 0;
    unsigned char *enc_buf = NULL;

    log_msg("\nbb_write(path=\"%s\", buf=0x%08x, size=%d, offset=%lld, fi=0x
        path, buf, size, offset, fi);
    // no need to get fpath on this one, since I work from fi->fh not the pa
    log_fi(fi);

    // encrypt data
    retstat = enc_encrypt_data((const unsigned char *)buf, size, &enc_buf);
    // if successful, pwrite()
    if (retstat == 0) {
        retstat = pwrite(fi->fh, enc_buf, size, offset);
        free(enc_buf);
    }

    if (retstat < 0)
        retstat = log_error("bb_write");

    return retstat;
}
```

Figure 2: bb_write(), encrypt data before pwrite()

```c
    // Pull the rootdir out of the argument list and save
    // internal data
    bb_data->rootdir = realpath(argv[argc-2], NULL);
    argv[argc-2] = argv[argc-1];
    argv[argc-1] = NULL;
    argc--;

    bb_data->logfile = log_open();
    enc_get_keys(&bb_data->key_add, &bb_data->key_shift);
```

Figure 3: main(), save add/shift keys in BB_DATA

```
▼ src
    bbfs.c
    ee516.conf
    encryption.c
    encryption.h
```

```c
#pragma once

#include <stdlib.h>

void enc_decrypt_data
(unsigned char *buf, size_t size);

int enc_encrypt_data
(const unsigned char *buf, size_t size, unsigned char **enc_buf);

void enc_get_keys
(unsigned int *add_key, unsigned int *shift_key);
```

Figure 4: APIs of encryption module (**encryption.h**)

```c
/* decrypt */
for (i = 0; i < size; i++)
{
    /* circular left shift */
    buf[i] = (buf[i] << key_shift) | (buf[i] >> (sizeof(unsigned char) * 8 - key_shift));

    /* subtract */
    buf[i] = buf[i] - key_add;
}
```

Figure 5: **enc_decrypt_data**() logic, circular left shift + subtraction

```c
/* encrypt */
for (i = 0; i < size; i++)
{
    /* add */
    (*enc_buf)[i] = (*enc_buf)[i] + key_add;

    /* circular right shift */
    (*enc_buf)[i] = ((*enc_buf)[i] >> key_shift) | ((*enc_buf)[i] << (sizeof(unsigned char) * 8 - key_shift));
}
```

Figure 6: **enc_encrypt_data**() logic, addition + circular right shift

```c
int matched;

// read keys
matched = fscanf(fp_conf, "%u %u", &_add_key, &_shift_key);

// ensure read correctly
if (matched != 2)
    _add_key = _shift_key = 0;
```

Figure 7: **enc_get_keys**() logic, reading add + shift keys from configuration file

```
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# ./bbfs -o direct_io /root/fuse/ /mnt/fuse/
about to call fuse_main
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# cat ee516.conf
1 2
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# cd /mnt/fuse/
root@gvkalra-desktop:/mnt/fuse# echo "Hello FUSE!" > hellofuse.txt
root@gvkalra-desktop:/mnt/fuse# cat hellofuse.txt
Hello FUSE!
root@gvkalra-desktop:/mnt/fuse# cd -
/home/gvkalra/Desktop/EE516/PR04/task01/src
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# fusermount -u /mnt/fuse/
```

Figure 8: Writing & Reading "Hello FUSE!" to hellofuse.txt

add_key (1) and shift_key (2)

```
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# echo "4 4" > ee516.conf
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# ./bbfs -o direct_io /root/fuse/ /mnt/fuse/
about to call fuse_main
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# cd -
/mnt/fuse
root@gvkalra-desktop:/mnt/fuse# cat hellofuse.txt
!±±½UM(root@gvkalra-desktop:/mnt/fuse# cd -
/home/gvkalra/Desktop/EE516/PR04/task01/src
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# fusermount -u /mnt/fuse/
```

Figure 9: Reading hellofuse.txt

add_key (4) and shift_key (4)

```
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# echo "1 2" > ee516.conf
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# ./bbfs -o direct_io /root/fuse/ /mnt/fuse/
about to call fuse_main
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# cd -
/mnt/fuse
root@gvkalra-desktop:/mnt/fuse# cat hellofuse.txt
Hello FUSE!
root@gvkalra-desktop:/mnt/fuse# cd -
/home/gvkalra/Desktop/EE516/PR04/task01/src
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task01/src# fusermount -u /mnt/fuse/
```

Figure 10: Reading hellofuse.txt

add_key (1) and shift_key (2)

Task 2: Buffer cache implementation on FUSE file system

```
int bb_read(const char *path, char *buf, size_t size, off_t of
{
    int retstat = 0;

    log_msg("\nbb_read(path=\"%s\", buf=0x%08x, size=%d, offse
        path, buf, size, offset, fi);
    // no need to get fpath on this one, since I work from fi-
    log_fi(fi);

    if (size != 4096) {
        log_msg("ERROR : size must be 4K");
        return -1;
    }

    // read from cache
    retstat = buf_read(fi->fh, buf, size, offset, fi->flags);
```

Figure 1: bb_read(), buffered read buf_read()

```
int bb_write(const char *path, const char *buf, size_t size, off_t offse
    struct fuse_file_info *fi)
{
    int retstat = 0;
    unsigned char *enc_buf = NULL;

    log_msg("\nbb_write(path=\"%s\", buf=0x%08x, size=%d, offset=%lld, f
        path, buf, size, offset, fi);
    // no need to get fpath on this one, since I work from fi->fh not th
    log_fi(fi);

    if (size != 4096) {
        log_msg("ERROR : size must be 4K");
        return -1;
    }

    // encrypt data
    retstat = enc_encrypt_data((const unsigned char *)buf, size, &enc_bu
    // if successful, buf_write()
    if (retstat == 0) {
        retstat = buf_write(fi->fh, enc_buf, size, offset, fi->flags);
        free(enc_buf);
    }
```

Figure 2: bb_write(), buffered write buf_write()

```
int bb_flush(const char *path, struct f
{
    int retstat = 0;

    log_msg("\nbb_flush(path=\"%s\", fi
    // no need to get fpath on this one
    log_fi(fi);

    retstat = buf_flush(fi->fh);
```

Figure 3: bb_flush(), flushing dirty data with buf_flush()

```
bb_data->logfile = log_open();
enc_get_keys(&bb_data->key_add, &bb_data->key_shift);
buf_get_policy(&bb_data->buf_policy);
```

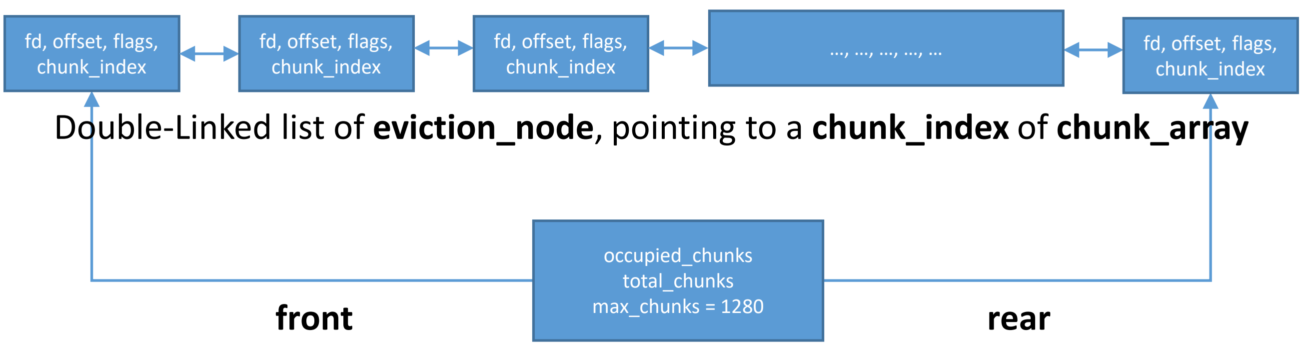
Figure 4: main(), reading/saving buffer policy in BB_DATA

```
buffer.h
#pragma once

#include <unistd.h>

void buf_get_policy(unsigned int *buf_policy);

ssize_t buf_read(int fd, void *buf, size_t count, off_t offset, int flags);

ssize_t buf_write(int fd, const void *buf, size_t count, off_t offset, int flags);

int buf_close(int fd);
int buf_flush(int fd);
```

Figure 5: APIs of cache buffer module (**buffer.h**)

Key Idea:

chunk_array consists of 1280 elements, each of which is 4KB in size



Double-Linked list of **eviction_node**, pointing to a **chunk_index** of **chunk_array**

eviction_queue

For each **bb_read() / bb_write()**:

1. Try reading (or writing) from (or to) cache

```
// check buffer
if (_tryread_cache(fd, buf, count, offset) == count) {
    log_msg("Cache HIT\n");
    return count;
}
```

```
// check buffer
if (_trywrite_cache(fd, buf, count, offset) == count) {
    log_msg("Cache HIT\n");
    return count;
}
```

2. If cache miss, try expanding **eviction_queue**

```
// expand queue if possible
if (is_evic_queue_expandable()) {
    log_msg("Expandable Cache\n");
    node = create_new_node(fd, offset, flags);
}
```

3. If queue is at it's maximum limit (1280), try utilizing an existing **eviction_node** which might have been flushed to disk, thereby available for re-utilization. In case there is no such node available, run eviction algorithm.

```
// buffer full, evict
else if (is_evic_queue_full()) {
    log_msg("Eviction Cache\n");
    node = _evict_cached_node(); //returns evicted node
}
// buffer available, reuse
else {
    log_msg("Re-usable Cache\n");
    node = _find_usable_node();
}
```

Task 3: File system performance evaluation

```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR04/task03 (master) $ ./task3_fsbench /home/gvkalra/Desktop/ 4096
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
=============  File System Benchmark Execution Result (Time usec)  =============
File Create       :            136
Sequential Write :           12776
Sequential Read  :            2624
Random Write      :            5741
Random Read       :            2428
File Delete       :            2434
Total             :           26139
================================================================================
```

Figure 1: Current Linux file system

```
ee516.conf
1  0 0
2  0
```

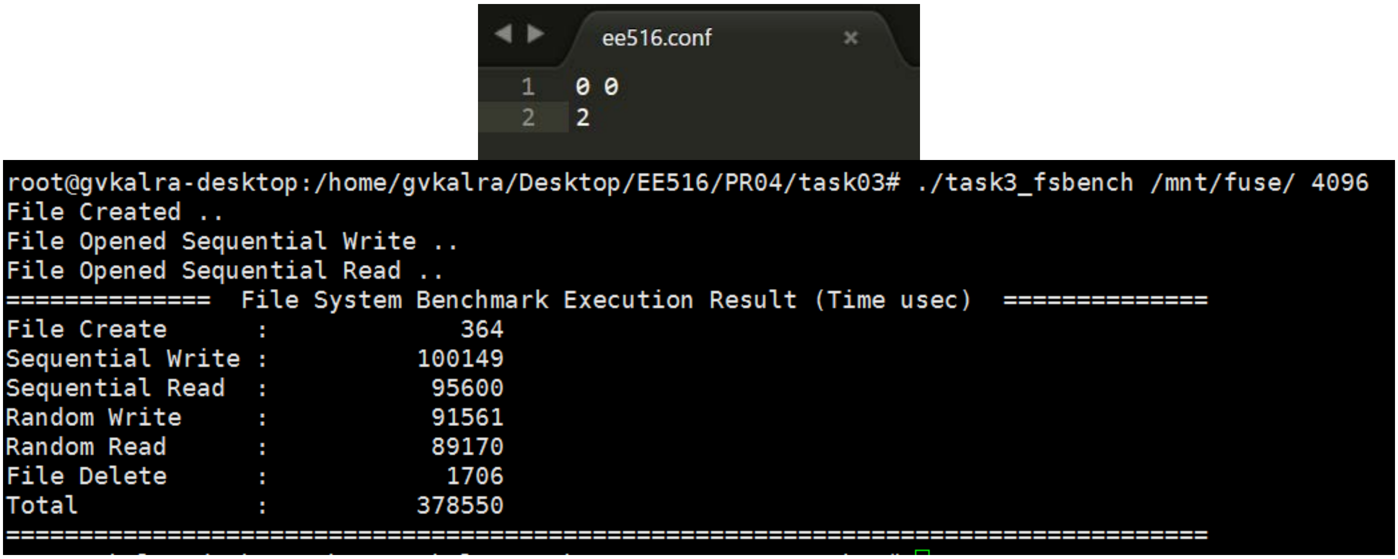
```
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task03# ./task3_fsbench /mnt/fuse/ 4096
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create       :            832
Sequential Write :           67955
Sequential Read  :           63717
Random Write      :           66874
Random Read       :           64351
File Delete       :            1516
Total             :          265245
==================================================================================
```

Figure 2: FUSE with **no encryption and no buffer**

```
ee516.conf
1  0 0
2  1
```

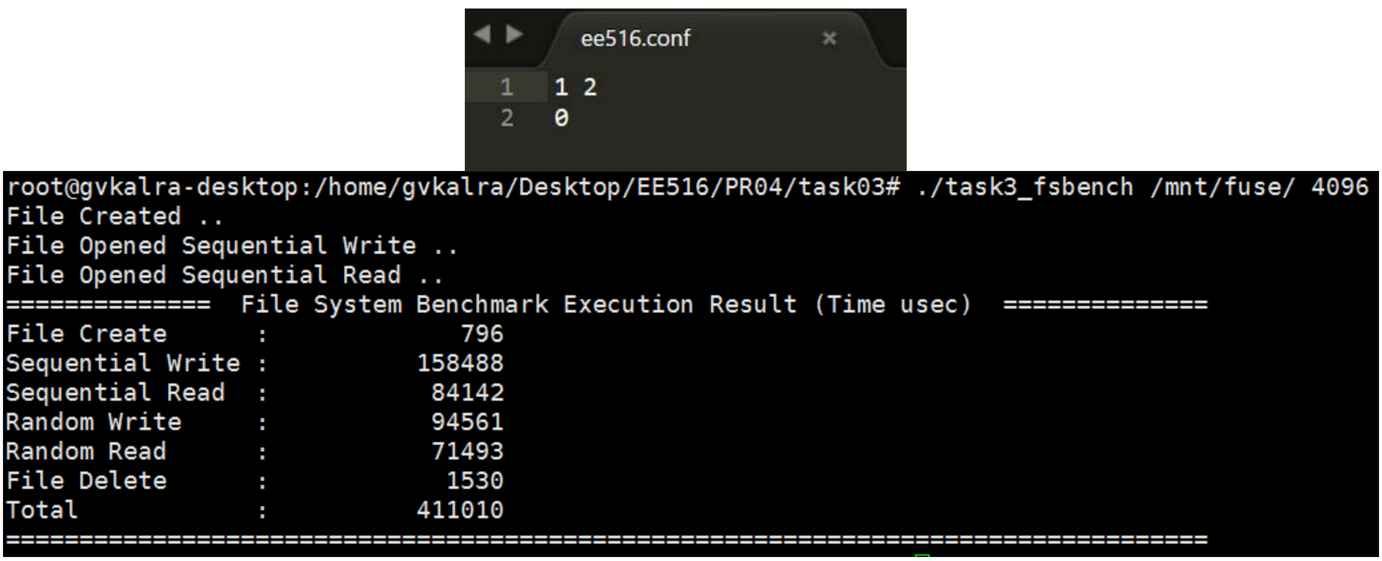
```
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task03# ./task3_fsbench /mnt/fuse/ 4096
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create       :            278
Sequential Write :           97334
Sequential Read  :           92236
Random Write      :           87185
Random Read       :           86572
File Delete       :            1837
Total             :          365442
==================================================================================
```

Figure 3: FUSE with **no encryption and random eviction buffer**



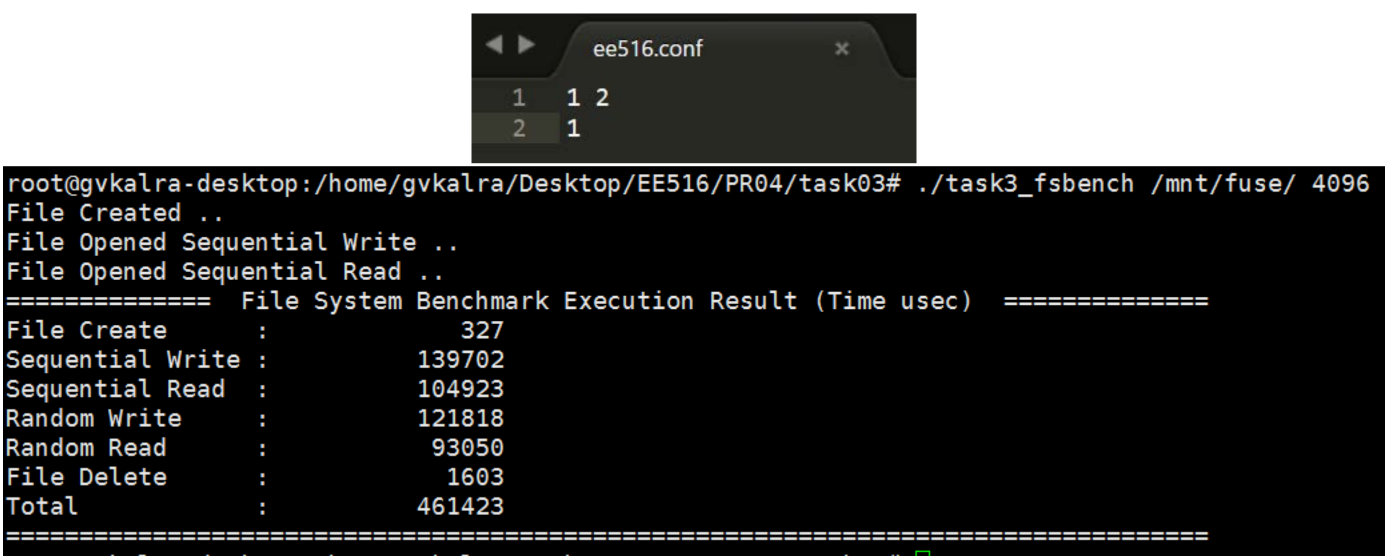
```
ee516.conf
1  0 0
2  2

root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task03# ./task3_fsbench /mnt/fuse/ 4096
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create       :          364
Sequential Write :       100149
Sequential Read  :        95600
Random Write      :        91561
Random Read       :        89170
File Delete       :         1706
Total             :       378550
===================================================================================
```

Figure 4: FUSE with **no encryption and LRU buffer**



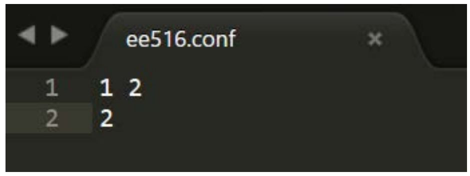
```
ee516.conf
1  1 2
2  0

root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task03# ./task3_fsbench /mnt/fuse/ 4096
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create       :          796
Sequential Write :       158488
Sequential Read  :        84142
Random Write      :        94561
Random Read       :        71493
File Delete       :         1530
Total             :       411010
===================================================================================
```

Figure 5: FUSE with **encryption and no buffer**



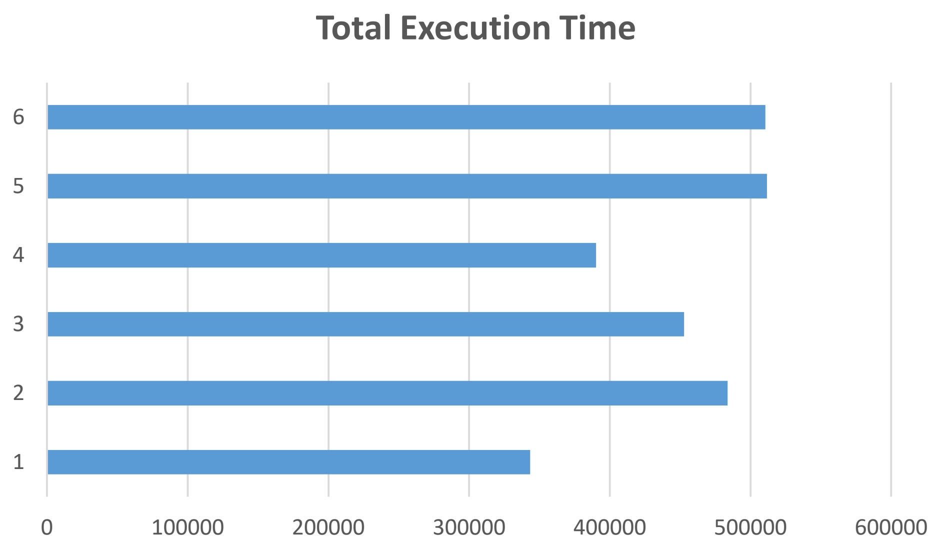
```
ee516.conf
1  1 2
2  1

root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task03# ./task3_fsbench /mnt/fuse/ 4096
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create       :          327
Sequential Write :       139702
Sequential Read  :       104923
Random Write      :       121818
Random Read       :        93050
File Delete       :         1603
Total             :       461423
===================================================================================
```

Figure 6: FUSE with **encryption and random eviction buffer**




```
root@gvkalra-desktop:/home/gvkalra/Desktop/EE516/PR04/task03# ./task3_fsbench /mnt/fuse/ 4096
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create      :          278
Sequential Write :       131374
Sequential Read  :       105012
Random Write     :       118379
Random Read      :       101619
File Delete      :         1686
Total            :       458348
==================================================================================
```

Figure 7: FUSE with **encryption and LRU buffer**



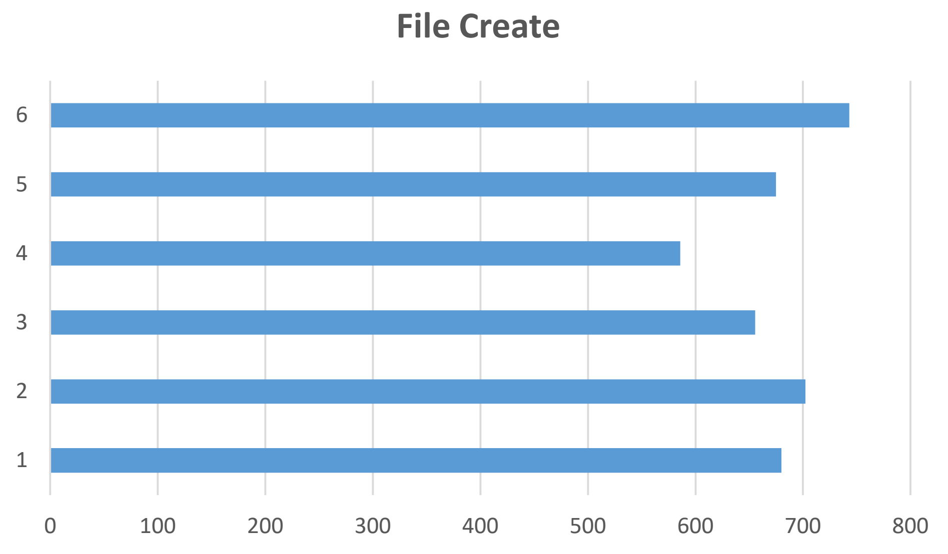
Legend

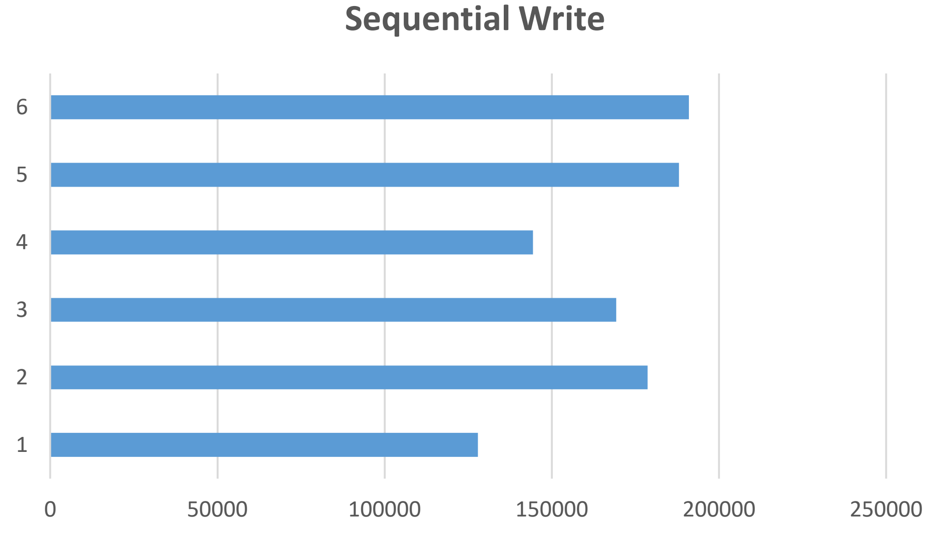
FUSE with:

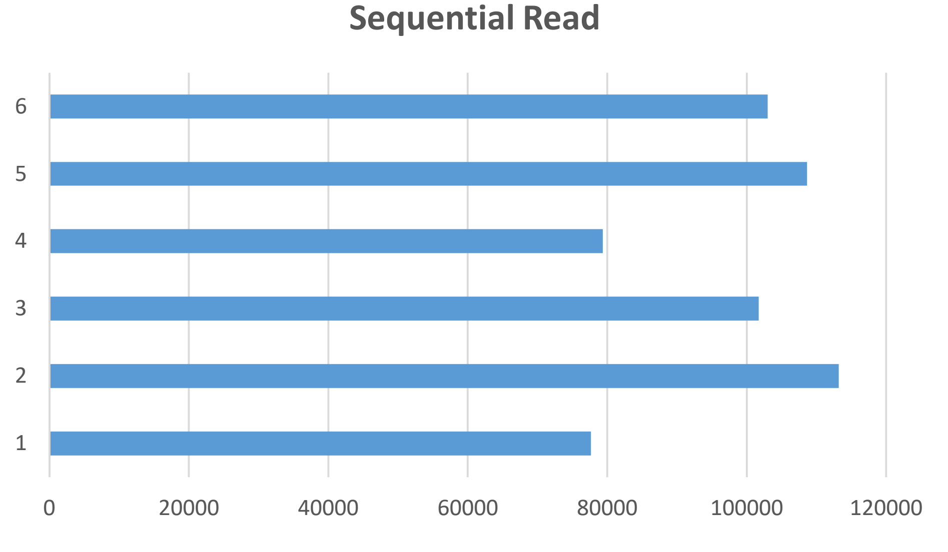
1. No encryption and no buffer
2. No encryption and random eviction buffer
3. No encryption and LRU buffer
4. Encryption and no buffer
5. Encryption and random eviction buffer
6. Encryption and LRU buffer

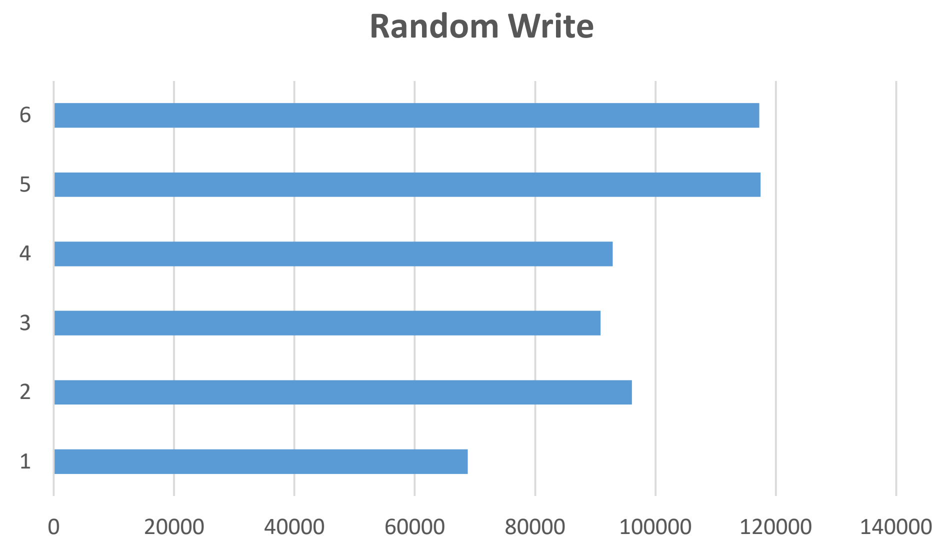
Notes:

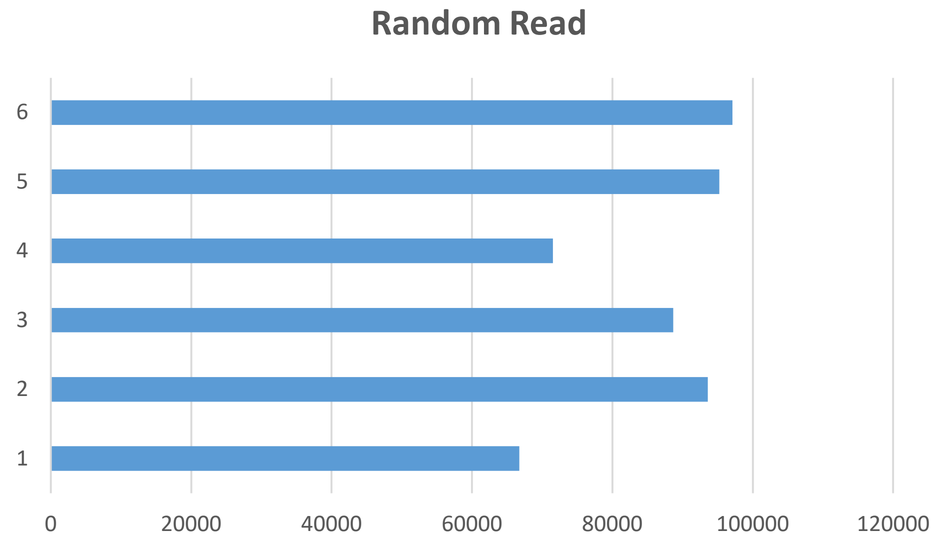
Time is in usec and data is averaged over 5 executions.

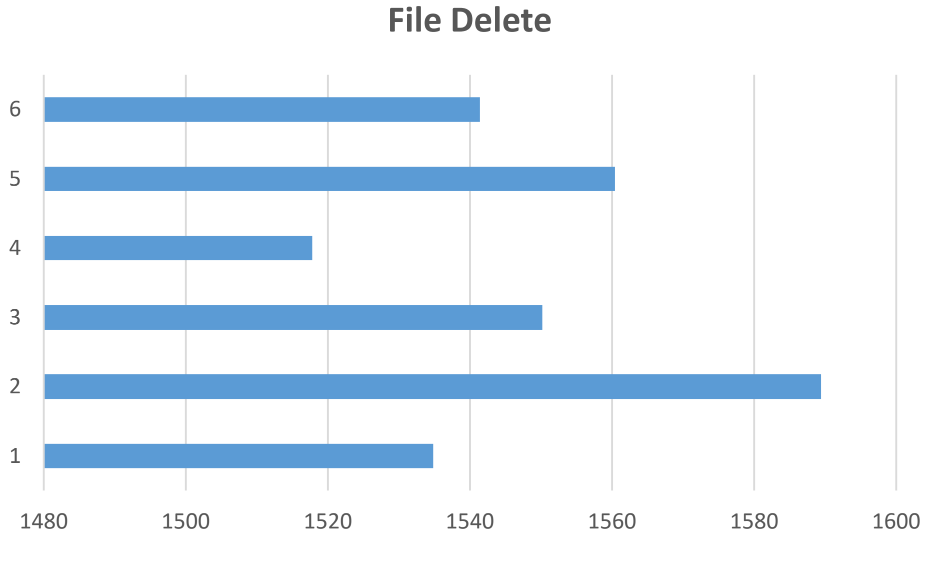

File Create
6
5
4
3
2
1
0
100
200
300
400
500
600
700
800


Sequential Write
6
5
4
3
2
1
0
50000
100000
150000
200000
250000


Sequential Read
6
5
4
3
2
1
0
20000
40000
60000
80000
100000
120000


Random Write
6
5
4
3
2
1
0
20000
40000
60000
80000
100000
120000
140000


Random Read
6
5
4
3
2
1
0
20000
40000
60000
80000
100000
120000


File Delete
6
5
4
3
2
1
1480
1500
1520
1540
1560
1580
1600

Discussion:

The total execution time has the following order:

No encryption and no buffer **(fastest)**

<

Encryption and no buffer

<

No encryption and LRU buffer

<

No encryption and random eviction buffer

<

Encryption and random eviction buffer

< (almost same)

Encryption and LRU buffer **(slowest)**

Why?

1. Any buffering algorithm is as good as the workload it targets. The benchmarking program doesn’t have a predefined workload which makes buffering an overhead.
2. The number of cache misses far outweigh cache hits in the benchmarking workload.

Q1. Pros. and Cons. of FUSE file system (against file system in kernel space)

Pros:

1. It lets non-privileged users create their own file systems without editing kernel code.
2. FUSE module provides only a "bridge" to the actual kernel interfaces.
3. It is particularly useful for writing virtual file systems. Unlike traditional file systems that essentially save data to, and retrieve data from, mass storage, virtual filesystems do not actually store data themselves. They act as a view or translation of an existing file system or storage device.
4. If a FUSE filesystem driver crashes, it won't panic your kernel: you'll see nothing worse than I/O errors in applications that were accessing the filesystem.
5. They can be programmed very quickly

Cons:

1. They're somewhat slower in comparison to file system in kernel space. This is mainly because of more context switches between user-space and kernel-space
2. It is not robust because a crashing / killed fuse process by mistake can take away the whole filesystem
3. They cannot be used on a boot media.

References:

https://en.wikipedia.org/wiki/Filesystem_in_Userspace
http://unix.stackexchange.com/a/4170

Q2. In our encryption, can different key pairs give same encrypted / decrypted data? Why?

Yes.

Encryption = Addition + Circular right shift
Decryption = Circular left shift + Subtraction

Let a data byte be 0xFF
Let one (add, shift) pair be (1, 2) and another be (1, 4)

1.1 Encryption

0xFF + 1 = 0x00 (since carry bit is lost in our encryption scheme)
$(0x00 >> 2)_{circular}$ = 0x00

1.2 Decryption

$(0x00 << 2)_{circular}$ = 0x00
0x00 - 1 = 0xFF

2.1 Encryption

0xFF + 1 = 0x00
$(0x00 >> 4)_{circular}$ = 0x00 (same as 1.1)

2.2 Decryption

$(0x00 << 4)_{circular}$ = 0x00
0x00 - 1 = 0xFF (same as 1.2)

Q3. List several encryption methods and analyze them.

To provide strong enough encryption it is necessary to encrypt as much data together in a chaining fashion that includes bit substitutions and transpositions, such that each byte encrypted depends on some of the prior ones. However, doing so would mean that each time we need to decrypt a single byte anywhere in the file, all prior bytes would have to be decrypted as well -- a major performance problem. So, in general, fixed-length block ciphers are used in file system encryption. e.g.

1. DES (too big and slow)
2. Blowfish (fast, compact, simple)

Cryptfs (A Stackable Vnode Level Encryption File System) uses Blowfish with Cipher Block Chaining (CBC) encryption mode for each block (4-8KB depending on page size) to be encrypted.

References:
http://www.fsl.cs.sunysb.edu/docs/cryptfs/node2.html#SECTION00022000000000000000
https://en.wikipedia.org/wiki/Blowfish_(cipher)
https://en.wikipedia.org/wiki/Data_Encryption_Standard
https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#CBC

Q4. Why performances of Linux and FUSE file systems are different?
A Linux file system (e.g. ext4) runs completely inside a kernel, whereas part of a FUSE file system executes within user-space. This means to perform an operation using Linux file system, the user space process needs to invoke a single system call, which gets handled by the implementation of VFS inside the kernel.

However, in case of FUSE file system, VFS delegates the handling responsibility to FUSE, which further delegates it to a user-space program. In brief, FUSE introduces an extra layer of context switching between user and kernel space, thus contributing to degraded performance.

References:
https://github.com/libfuse/libfuse#about

Q5. Why performances of different eviction algorithms are different? In which cases each eviction algorithm can have advantage?
Performance of an eviction algorithm depends on the locality of workload (data to read and write). That is why performance of different eviction algorithms are different under various scenarios and there is no one eviction algorithm suitable for all types of workloads. e.g.

Least Recently Used:
It evicts the least recently used victim. In other words, LRU responds quickly to what has happened recently. This is advantageous for most types of user activities. However, this general prediction may not be valid for all types of workloads. e.g although LRU responds to repeated requests quickly, it gets burdened by long scans since we need to maintain aging information.

Random Eviction:
It chooses the victim of eviction randomly. In other words, it doesn't need to maintain aging information (unlike LRU). However, random workloads occur rarely in reality. Most memory, filesystem related workloads have a locality pattern and are not random. Random eviction may be better suited when there is no inherent information associated with access patterns, which is seldom (rare).