4η Εργασία Λειτουργικών Συστημάτων

Φοιτητές: Γρηγόρης Ηλιάδης, Δημήτρης Κεφαλάς, Φοίβος Πουρναρόπουλος
ΑΕΜ: 2522, 2533, 2614 E-mail: giliadis@uth.gr, dkefalas@uth.gr, spournar@uth.gr
Διδάσκων: Εμμανουήλ Κουτσουμπέλιας
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Ηλεκτρονικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας

Στην 4η εργασία καλούμαστε να υλοποιήσουμε ένα σύστημα αρχείων με βάση την έκδοση FUSE Big Brother File System (BBFS), το οποίο προσπαθεί να ελαχιστοποιήσει τον αριθμό των απαιτούμενων blocks που χρειάζονται τα αρχεία ώστε να αποθηκευτούν στο δίσκο, εντοπίζοντας και επαναχρησιμοποιώντας κοινά blocks αρχείων με όμοιο περιεχόμενο.

Από τις παραδοχές-απλουστεύσεις που υπάρχουν στην εκφώνηση ακολουθούμε τις εξής:

• Αν τυχόν δύο blocks παράγουν το ίδιο hash, μπορείτε να υποθέσετε ότι είναι ίδια. Στην πραγματικότητα, με δεδομένο ότι ρεαλιστικά το hash αποτελείται από πολύ λιγότερα bits σε σχέση με το μέγεθος του block, περισσότερα από ένα διαφορετικά στιγμιότυπα περιεχομένου είναι δυνατόν να οδηγήσουν στο ίδιο hash. Το φαινόμενο αυτό λέγεται hash collision. Στα πλαίσια της εργασίας θα το αγνοήσουμε.

• Μπορείτε να θεωρήσετε ότι ένα μόνο πρόγραμμα (και native thread) προσπελαύνει το σύστημα αρχείων κάθε χρονική στιγμή. Δεν απαιτείται δηλαδή να ασχοληθείτε με ζητήματα thread safety της υλοποίησης του filesystem.

Οι επιπλέον λειτουργίες που έχουν υλοποιηθεί είναι οι εξής:

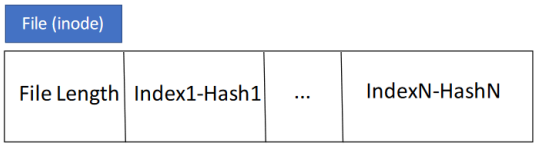
• Οι αιτήσεις ανάγνωσης και εγγραφής γίνονται σε οποιαδήποτε απόσταση από την αρχή του αρχείου χωρίς να είναι απαραίτητα πολλαπλάσια των 4KB. Δεν υπάρχει κάποιος περιορισμός σχετικά με το μέγεθος του αρχείου.

• Υποστηρίζεται και μία επιπλέον λειτουργία εκτός από την ανάγνωση, την εγγραφή, δημιουργία και διαγραφή αρχείων, η truncate.

• Το σύστημα αρχείων μπορεί να φιλοξενεί όσα αρχεία επιθυμεί ο χρήστης, χωρίς να υπάρχει κάποιος περιορισμός για το μέγεθος τους.

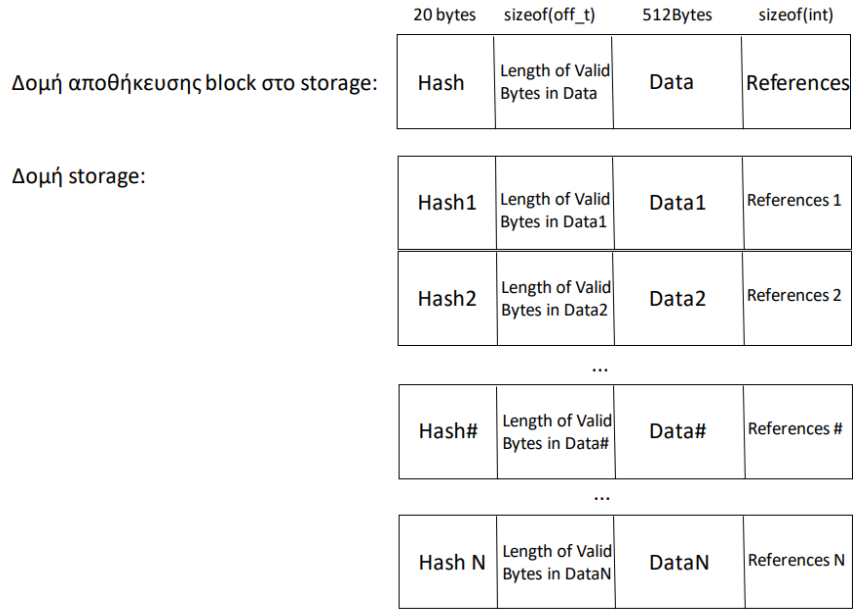
• Το σύστημα μας είναι non volatile, δηλαδή ο χρήστης μπορεί να κάνει unmount και μετέπειτα mount διατηρώντας τα αρχεία που είχαν αποθηκευτεί πριν στο σύστημα αρχείων.

Αρχιτεκτονική Συστήματος Αρχείων: Αντιμετωπίζουμε κάθε εικονικό μας αρχείο ως το inode του αντίστοιχου αρχείου στο filesystem. Τα περιεχόμενα του είναι το συνολικό μέγεθος του αρχείου και οι αναφορές-δείκτες στα blocks του αρχείου στην αποθήκη.



Εικόνα (1.1) Δομή αποθήκευσης του αρχείου στη μνήμη

Η αποθήκη μας υλοποιείται μέσα σε ένα κρυφό αρχείο που βρίσκεται στον φάκελο rootdir με όνομα .storage, στο οποίο αποθηκεύονται τα blocks του κάθε αρχείου μαζί με επιπλέον πληροφορία που χρησιμοποιείται για την διαχείριση της αποθήκης. Τα μεταδεδομένα που αποθηκεύονται είναι το hash code του κάθε block, το μέγεθος των έγκυρων δεδομένων του block, τα δεδομένα του κάθε block, όπως επίσης και τον αριθμό των αναφορών στο συγκεκριμένο block από οποιοδήποτε αρχείο στο σύστημα αρχείων μας. Το κάθε block στην αποθήκη έχει μοναδικό hash code, έτσι εξοικονομούμε χώρο για την αποθήκευση των αρχείων στο δίσκο αφού κοινά blocks μεταξύ αρχείων ή ακόμα και του ίδιου αρχείου, παράγουν το ίδιο hash code. Το μέγεθος του πεδίου Data είναι σταθερό και ίσο με 512 bytes. Εφόσον το συνολικό μέγεθος των αρχείων δεν είναι συγκεκριμένο, αυτό έχει ως αποτέλεσμα κατά τη διάσπαση του αρχείου, να προκύψει block το οποίο έχει μέγεθος δεδομένων μικρότερο των 512 bytes. Για το λόγο αυτό αποθηκεύουμε και το μέγεθος των έγκυρων δεδομένων του block. Εφόσον το κάθε block χρησιμοποιείται από ένα ή περισσότερα αρχεία, το πεδίο references (αριθμός των αναφορών στο συγκεκριμένο block) δείχνει πόσα αντίγραφα του συγκεκριμένου block θα υπήρχαν στο δίσκο σε ένα συμβατικό σύστημα αρχείων. Ένα block διαγράφεται από την αποθήκη μας μόνο όταν δεν υπάρχει καμία αναφορά σε αυτό (references = 0).



Εικόνα 1.2
Δομή οργάνωσης block και storage

Η επιπλέον πληροφορία που αποθηκεύουμε αντιστοιχεί στο 6.25% του συνολικού μπλοκ. Εάν θέλουμε να ελαττώσουμε το συνολικό overhead των metadata του κάθε block που προστίθεται της αποθήκης μας, μπορούμε να αυξήσουμε το μέγεθος των δεδομένων από 512 bytes. Ενδεικτικό overhead για

dataSize = 1024 -> overhead = 3.1250%

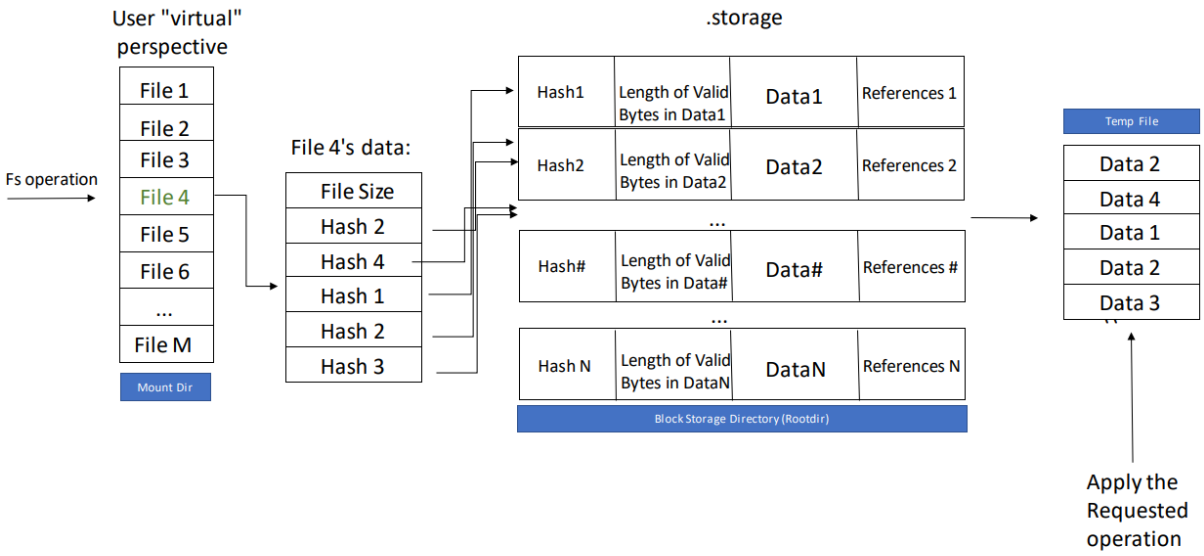
dataSize = 2048 -> overhead = 1.5625%

dataSize = 4096 -> overhead = 0.7812%.

Αυξάνοντας όμως το μέγεθος των δεδομένων του block, μειώνουμε την πιθανότητα να βρεθούνε δύο ίδια blocks στο filesystem μας. Επειδή ο στόχος μας είναι να εξοικοινομήσουμε χώρο στο δίσκο επιλέξαμε

το μέγεθος δεδομένων του block να είναι 512 bytes, έχοντας μεγαλύτερη πιθανότητα ύπαρξης κοινών block στα αρχεία του συστήματος αρχείων μας.

Η λογική της αρχιτεκτονικής του συστήματος αρχείων είναι η εξής: Στον φάκελο που κάναμε mount το σύστημα αρχείων μας, περιέχονται αρχεία τα οποία έχουν για δεδομένα το μέγεθος που θα είχαν τα αρχεία σε ένα συμβατικό σύστημα αρχείων καθώς και τα hash codes των block τους. Όταν μία λειτουργία του συστήματος αρχείων γίνεται σε ένα αρχείου του, τότε μέσω του εικονικού αρχείου βρίσκουμε τα block του πραγματικού αρχείου στο δίσκο με βάση το hash code, και αναδημιουργούμε το πραγματικό αρχείο σε ένα προσωρινό. Εκτελούμε οποιαδήποτε ενέργεια χρειαστεί στο προσωρινό αρχείο και έπειτα το ξαναχωρίζουμε σε blocks που αποθηκεύονται πίσω στο δίσκο, ενημερώνουμε το εικονικό αρχείο με τα νέα hash codes και file size εφόσον χρειάζεται. Τέλος, διαγράφουμε από την αποθήκη όσα blocks που πλέον δεν χρησιμοποιούνται.



Εικόνα 1.3
Αρχιτεκτονική Συστήματος Αρχείων

Για παράδειγμα, έστω ότι ο χρήστης θέλει να κάνει μία συγκεκριμένη λειτουργία σε ένα αρχείο του Συστήματος Αρχείων ας πούμε το αρχείο File 4. Τότε το Σύστημα Αρχείων ανοίγει το αρχείο FILE4 για ανάγνωση και εγγραφή, διαβάζει το πραγματικό μέγεθος του (File4 size) και στην συνέχεια τα hashes των block του (hash2, hash4, hash1, hash2, hash3). Μετέπειτα με βάση το hash του κάθε block, ανακτούνται τα δεδομένα του (hash2->data2, hash4->data4, hash1->data1, hash2->data2, hash3->data3) και εγγράφονται σε ένα προσωρινό αρχείο (περιεχόμενα προσωρινού αρχείου: data2, data4, data1, data2, data3). Στο προσωρινό αρχείο πραγματοποιείται η αντίστοιχη λειτουργία του εικονικού αρχείου με το ίδιο τρόπο που θα γινόταν σε ένα συμβατικό σύστημα αρχείων. Εάν αυτή η λειτουργία έχει προσθέσει, διαγράψει ή ακόμα και μετατρέψει προυπάρχουσα δεδομένα του προσωρινού αρχείου, το αρχείο διασπάται εκ νέου σε block, το κάθε block αντιστοιχείται με ένα hash όπου τοποθετείται στην αποθήκη αλλά και στο εικονικό αρχείο για να λειτουργεί ως δείκτης προς το συγκεκριμένο block στο storage. Προκειμένου να καθαριστεί η αποθήκη από blocks που δεν χρησιμοποιούνται πλέον, έχει υλοποιηθεί ένας garbage collector που ελέγχει την αποθήκη μετά από κάθε προσπέλαση της.

Για να υλοποιήσουμε την παραπάνω λειτουργικότητα, χρειάστηκε να αλλάξουμε τις εξής συναρτήσεις του bbfs.

Κατάλογος συναρτήσεων του BBFS που χρειάστηκε να αλλάξουμε:

1)bb_getattr: Στην υλοποίηση του BBFS στην bb_getattr καλείται η συνάρτηση lstat προκειμένου να δώσει τιμές σχετικά με διάφορες πληροφορίες του αρχείου σε ένα struct stat που δίνεται ως παράμετρος. Μια από αυτές τις πληροφορίες είναι και το μέγεθος του αρχείου. Επειδή το αρχείο που βλέπει το λειτουργικό είναι το εικονικό, η τιμή που θα δοθεί από την κλήση της lstat, στο πεδίο st_size του struct stat δίνεται το μέγεθος του εικονικού αρχείου. Επειδή η bb_getattr πρέπει να επιστρέφει το πραγματικό μέγεθος του αρχείου ανανεώνουμε το πεδίο st_size του struct ως εξής: ανοίγουμε το εικονικό αρχείο (μόνο για ανάγνωση). Στα πρώτα #off_t bytes έχει αποθηκευτεί το πραγματικό μέγεθος του αρχείου, έτσι το μόνο που χρειάζεται είναι μία ανάγνωση των πρώτων #off_t bytes του εικονικού αρχείου για να πάρουμε αυτή την πληροφορία.

2)bb_fgetattr: Επειδή στα πειράματα που κάναμε χρησιμοποιήσαμε εντολές του shell (πχ cat, diff κλπ), παρατηρήσαμε ότι αντί για την bb_getattr καλούταν η bb_fgetattr, έτσι αλλάξαμε και την bb_fgetattr ακολουθώντας την ίδια φιλοσοφία με την bb_getattr.

3)bb_unlink: Σκοπός της bb_unlink είναι η διαγραφή ενός αρχείου, το filepath του οποίου δίνεται ως παράμετρος σε αυτή. Αυτό το filepath αντιστοιχεί στο εικονικό αρχείο, τα περιεχόμενα του οποίου είναι αναφορές των block του στο δίσκο. Επειδή θέλουμε να διαγράψουμε και τα πραγματικά δεδομένα του, αναζητούμε τα blocks του αρχείου προς διαγραφή στην αποθήκη με βάση τα hash codes τους και μειώνουμε την τιμή των αναφορών σε αυτά κατά 1. Εάν οι αναφορές για κάποιο από αυτά είναι ίσες με το μηδέν (κανένα αρχείο δεν χρησιμοποιεί το συγκεκριμένο block) τότε τη θέση του συγκεκριμένου block παίρνει το τελευταίο block της αποθήκης και κατά συνέπεια μειώνεται ο χώρος της αποθήκης κατά το μέγεθος ενός block (Εικόνα 1.1). Σε αντίθετη περίπτωση, όπου οι αναφορές είναι θετικός αριθμός, το block παραμένει στην αποθήκη καθώς χρησιμοποιείται από άλλα αρχεία στο σύστημά μας.

4)bb_truncate: Δημιουργούμε ένα βοηθητικό προσωρινό αρχείο όπου περιέχει τα δεδομένα του αρχείου που θέλουμε να γίνει truncate και καλούμε την ftruncate για αυτό. Στη συνέχεια, υπολογίζουμε το νέο μέγεθος του, το χωρίζουμε σε blocks, βρίσκουμε τα νέα hash codes των blocks κάνουμε τις απαραίτητες ενέργειες για την αποθήκευση των block στην αποθήκη και ενημερώνουμε το εικονικό αρχείο με το νέο μέγεθος και τα hash codes των block του. Τέλος, χρησιμοποιούμε τον garbage_collector, σε περίπτωση που κάποιο block του αρχικού αρχείου δεν χρειάζεται να υπάρχει πλέον στην αποθήκη.

5)bb_ftruncate: Επειδή στα πειράματα που κάναμε χρησιμοποιήσαμε την εντολή truncate του shell, παρατηρήσαμε ότι αντί για την bb_truncate καλούταν η bb_ftruncate, έτσι αλλάξαμε και την bb_ftruncate ακολουθώντας την ίδια φιλοσοφία με την bb_truncate.

6)bb_open: Στην bb_open, ανοίγουμε το εικονικό αρχείο που ζητάει ο χρήστης δίνοντας δικαιώματα για ανάγνωση και εγγραφή. Θεωρούμε ότι σε κάθε προσπέλαση του εικονικού αρχείου υπάρχει πιθανότητα αλλαγής των δεδομένων του. Εάν το όνομά που δίνεται ως παράμετρος είναι ίδιο με το όνομα του αρχείου που υλοποιεί την αποθήκη ή με το όνομα του βοηθητικού προσωρινού αρχείου, η bb_open αποτυγχάνει.

5)bb_read: Υπολογίζουμε το συνολικό αριθμό των blocks του αρχείου και βρίσκουμε εκείνο από το οποίο θέλουμε να ξεκινήσει η ανάγνωση. Για όσα blocks χρειαστεί, διαβάζουμε τα δεδομένα τους και τα αποθηκεύουμε στον buffer που δίνεται ως παράμετρος. Έπειτα, επιστρέφουμε τον αριθμό των bytes που διαβάστηκαν.

6)bb_write: Δημιουργούμε ένα βοηθητικό προσωρινό αρχείο όπου περιέχει τα δεδομένα του αρχείου που θέλουμε να γίνει write και καλούμε την pwrite για αυτό. Στη συνέχεια, υπολογίζουμε το νέο μέγεθος του, το χωρίζουμε σε blocks, βρίσκουμε τα νέα hash codes των blocks κάνουμε τις απαραίτητες ενέργειες για την αποθήκευση των block στην αποθήκη και ενημερώνουμε το εικονικό αρχείο με το νέο μέγεθος και τα hash codes των block του. Τέλος, χρησιμοποιούμε τον garbage_collector, σε περίπτωση που κάποιο block του αρχικού αρχείου δεν χρειάζεται να υπάρχει πλέον στην αποθήκη.

7)bb_readdir: Η συνάρτηση δεν επιτρέπει στο χρήστη να «δει» το αρχείο της αποθήκης καθώς και το προσωρινό αρχείο της υλοποίησης.

8)bb_init: Δημιουργούμε το κρυφό αρχείο της αποθήκης (σε περίπτωση που δεν υπάρχει) καθώς και το προσωρινό αρχείο, αλλιώς τα ανοίγουμε κρατώντας τα παλιά δεδομένα του storage εφόσον το σύστημα αρχείων που υλοποιήσαμε είναι non-volatile.

9)bb_destroy:Κλείνουμε το αρχείο της αποθήκης και το προσωρινό αρχείο.

Σημείωση για την υλοποίηση: Για να μπορούμε να βρίσκουμε το κάθε block στην αποθήκη χρησιμοποιήσαμε την συνάρτηση find_block, η οποία αναζητά εάν υπάρχει block στην αποθήκη με hash code ίσο με το hash που δίνεται ως παράμετρος. Αυτό μας διευκολύνει σε σχέση με την περίπτωση όπου για κάθε block αποθηκεύαμε τη θέση του στο storage, καθώς μπορεί το ίδιο block να αλλάξει θέση στην αποθήκη, έπειτα από συγκεκριμένες λειτουργίες του συστήματος αρχείων. Αυτό θα απαιτούσε συνεχή ανανέωση της τιμής του εικονικού αρχείου, που αποθηκεύει τη θέση του block στο δίσκο. Στην περίπτωσή μας, αρκεί να αποθηκεύσουμε μία φορά την τιμή του hash code ενός block στο εικονικό αρχείο. Έτσι, εντοπίζουμε το κάθε block, ανεξάρτητα της θέσης του στην αποθήκη.

Έλεγχος λειτουργικότητας / ορθότητας

Για τον έλεγχο της ορθότητας έχει γραφτεί ένα script το οποίο έχει την ακόλουθη λειτουργία. Αρχικά αντιγράφονται δύο αρχεία κειμένου με την εντολή cp (τα αρχεία: test_files/test_8kb.txt, test_files/test_8kb_1.txt), με πολύ παρόμοιο περιεχόμενο στον mountdir. Για την ακρίβεια πρόκειται για το ίδιο κείμενο και στα δύο αρχεία, με τη διαφορά ότι στο δεύτερο αρχείο μερικοί χαρακτήρες είναι με κεφαλαία αντί με μικρούς που είναι στο αρχικό. Η διαδικασία της αντιγραφής ελέγχεται μέσω της εντολής diff που συγκρίνει τα αντίστοιχα αρχεία στο test_files και στο mountdir. Προκειμένου να ελέγξουμε εάν τα μεγέθη αρχείων είναι σωστά, συγκρίνουμε τα αντίστοιχα μεγέθη των αρχείων στον mountidir και στον test_files μέσω της εντολής stat και της σύγκρισης μεταξύ τους. Στη συνέχεια ακολουθεί ο έλεγχος σχετικά με το μέγεθος της αποθήκης. Μέσω της εντολής cmp βρίσκουμε ποια byte των αρχείων είναι διαφορετικά. Δεδομένου ότι το block που ανήκει ένα byte μπορεί να βρεθεί μέσω της εξής φόρμουλας byte's block = byte_offset_from_start/BLOCK_SIZE (ακέραια διαίρεση), συγκρίνοντας σε ποιο block βρίσκονται δύο συνεχόμενα διαφορετικά byte, υπολογίζουμε τον συνολικό αριθμό των block που διαφέρουν μεταξύ των δύο αρχείων. Επειδή τα αρχεία έχουν ίδιο μέγεθος, ο αριθμός των συνολικών block που είναι στην αποθήκη, δεδομένου ότι υπάρχουν μόνο αυτά τα 2 αρχεία, θα είναι ίσος με το αριθμό των block του ενός από τα δύο αρχεία συν τον αριθμό των block που διαφέρουν. Το μέγεθος της αποθήκης θα είναι ίσο με το γινόμενο των συνολικών block της επί το μέγεθος που καταλαμβάνει το κάθε block στην αποθήκη (εικόνα 1.1). Πραγματοποιείται και έλεγχος για την επιπρόσθετη λειτουργία της truncate, όπου ελαττώνουμε το μέγεθος και των δύο αρχεία αρχικά στα 4,5kb (το σύστημα αρχείων μας υποστηρίζει αρχεία ανεξαρτήτου μεγέθους όπως αναφέρεται πιο πάνω) ελέγχοντας εκ νέου το μέγεθος του αρχείου, τα περιεχόμενα του αλλά και το μέγεθος της αποθήκης. Ο έλεγχος της truncate επαναλαμβάνεται και για τελικό μέγεθος των αρχείων ίσο με το 0.

Ανάλυση αποτελεσμάτων

Με την εκτέλεση του script λάβαμε τα εξής αποτελέσματα:

Για το πρώτο στάδιο όπου τα αρχεία έχουν την αρχική τους μορφή:

FILESYSTEM	FILE1	FILE2	SIZE FILE1	SIZE FILE2	STORAGE SIZE	TOTAL MEMORY ALLOC
ΣΥΜΒΑΤΙΚΟ	Test_8kb.txt	Test_8kb_1.txt	8192bytes	8192BYTES	0	16384
OUR_BBFS	Test_8kb.txt	Test_8kb_1.txt	328bytes	328bytes	11424	12080

Παρατηρούμε μείωση του αποθηκευτικού χώρου που χρειάζεται το Filesystem για την αποθήκευση των block των αρχείων και τη σωστή διαπέραση τους κατά 27%.

Για το δεύτερο στάδιο όπου τα αρχεία έχουν γίνει truncate:

FILESYSTEM	FILE1	FILE2	SIZE FILE1	SIZE FILE2	STORAGE SIZE	TOTAL MEMORY ALLOC
ΣΥΜΒΑΤΙΚΟ	Test_8kb.txt	Test_8kb_1.txt	4608bytes	4608bytes	0	9216
OUR_BBFS	Test_8kb.txt	Test_8kb_1.txt	188bytes	188bytes	5984	6360

Παρατηρούμε μείωση του αποθηκευτικού χώρου που χρειάζεται το Filesystem για την αποθήκευση των block των αρχείων και τη σωστή διαπέραση τους κατά 31%.

Για την βέλτιστη περίπτωση, δύο όμοιων αρχείων:

FILESYSTEM	FILE1	FILE2	SIZE FILE1	SIZE FILE2	STORAGE SIZE	TOTAL MEMORY ALLOC
ΣΥΜΒΑΤΙΚΟ	Test_8kb.txt	Test_8kb(1).txt	8192bytes	8192BYTES	0	16384
OUR_BBFS	Test_8kb.txt	Test_8kb(1).txt	328bytes	328bytes	8704	9360

Παρατηρούμε μείωση του αποθηκευτικού χώρου που χρειάζεται το Filesystem για την αποθήκευση των block των αρχείων και τη σωστή διαπέραση τους κατά 43%.

Για την χειρότερη περίπτωση, δύο εντελώς διαφορετικών αρχείων:

FILESYSTEM	FILE1	FILE2	SIZE FILE1	SIZE FILE2	STORAGE SIZE	TOTAL MEMORY ALLOC
ΣΥΜΒΑΤΙΚΟ	Test_8kb.txt	Random_Text_8kb.txt	8192bytes	8192BYTES	0	16384
OUR_BBFS	Test_8kb.txt	Random_Text_8kb.txt	328bytes	328bytes	17408	18064

Παρατηρούμε αύξηση του αποθηκευτικού χώρου που χρειάζεται το Filesystem για την αποθήκευση των block των αρχείων και τη σωστή διαπέραση τους κατά 10%.

Παρατηρήσεις: Για αρχεία μεγαλύτερα των 500kilobytes, το filesystem μας είναι υπερβολικά αργό. Μικρή βελτίωση στο χρόνο επιφέρει η αύξηση του block σε μία μεγαλύτερη τιμή της τάξης των 4kb.

Στα πειράματα υπάρχει και ένα δεύτερο script, το test_2.sh που ελέγχει την non volatile λειτουργία του συστήματος αρχείων. Αρχικά ελέγχει εάν ο φάκελος mountdir, έχει γίνει mount στο σύστημα αρχείων μας, εάν δεν είναι γίνεται mount, και στη συνέχεια κα στις δύο περιπτώσεις διαγράφονται όλα τα δεδομένα του και γίνονται cp τα αρχεία του φακέλου test_files σε αυτόν. Έπειτα γίνεται unmount και ξανά mount όπου ελέγχουμε εάν τα αρχεία πριν το umount, υπάρχουν στον mountdir μέσω της ls και της diff.

Οδηγίες για την μεταγλώττιση και τη εκτέλεση των test βρίσκονται στο README.txt στο φάκελο experiments.