Rapport du projet PNL

Participants :

- Siyuan CHEN 21101435
- Runlin ZHOU 28717281

1.1 Mounting and testing a vanilla OcuicheFS

Installation du moduel OuicheFS :

1. Changement de la commande pour démarrer le VM QEMU

   ```
   make mkfs.ouichefs              # complier l'exécutable de partipation
   make img                        # initilisation de l'image
   ```

   Dans le scripte, on ajoute une variable : `HDC="-drive file=test.img,format=raw"`, ce qui peut ajouter un driver de disque

2. Installer le module OuicheFS

   ```
   # à l'extérieur de VM
   make KERNELDIR=$(uname -r)       # compilier le moduel

   # à l'intérieur de VM
   insmod ouichefs.ko
   mkdir -r /mnt/ouichefs           # créer un point de mountage
   mount /dev/sdc /mnt/ouichefs     # mouter l'image vers la destination
   ```

3. effectuer les opérations de read/write
   - par les commandes : cat / echo ...
   - par les tests : `gcc -o benchmark benchmark.c && ./benchmark`

1.2 Implementation of a benchmark

Pour vérifier la validité de notre implementation, on crée une série de fonction pour l'examiner. Dans notre ficher du test, nous offrons trois opérations pour manipuler le ficher (sous le répertoire /mnt/ouichefs) :

- duplication : en paramétré de deux arguments : le chemin du ficher source et du ficher cible. La fonction peut copier le contenu ficher source vers la destinataire.
- écriture : en paramétré de trois arguments : la message en attende de l'écrir, le chemin du ficher, la position que nous souhaitons de le placer.
- lecture : en paramétré de un seul argument : le chemin du ficher. La fonction peut présenter le contenu du ficher vers le stdin.

Par conséquent, le fichier de test utilise la fonction *duplication* pour dupliquer le fichier source sous */share* vers */mnt/ouichefs*. Après une série d'opérations sur le fichier dupliquant, nous pouvons

utiliser les commandes *ioctl* pour obtenir l'état actuel du fichier et la commande *diff* pour confirmer que le contenu du fichier est conforme au fichier source.

1.3 Reimplementation of the read and the write functions

En besoin de redéfinir une méthode de la lecture et l'écriture, nous rajoutons deux champs dans la structure file_operations. Voici la structure ouichefs_file_ops après la modification :

```
const struct file_operations ouichefs_file_ops = {
    // les autres opéraions
    .read = ouichefs_read,
    .write = ouichefs_write,
    .unlocked_ioctl = ouichefs_ioctl,
    // les autres opérations
};
```

La fonction ouichefs_read(struct file *file, char __user *data, size_t len, loff_t *pos) :

- Nous ajoutons une condition du if : `if (*pos >= file->f_inode->i_size)` pour indiquer que la lecture de ficher est arrivée à la fin. Dans ce cas, la fonction va retourne 0.
- En utilisent `*pos / OUICHEFS_BLOCK_SIZE` et `*pos % OUICHEFS_BLOCK_SZIE` , read() peut savoir à partir de quel octet de quel bloc que la lecture doit commencer.
- Le programme lira alors le contenu du bloc correspondant et le copiera dans l'espace utilisateur par la fonction copy_to_user()

  ```
  // ci est le ouichefs_inode_info
  // sb est le superblock de ce ficher : file->f_inode->i_sb;
  struct buffer_head *bh_index = sb_bread(sb, ci->index_block);
  struct ouichefs_file_index_block * index
          = (struct ouichefs_file_index_block *)bh_index->b_data;
  // iblock = *pos / OUICHEFS_BLOCK_SIZE
  int bno = index->blocks[iblock];
  struct buffer_head *bh = sb_bread(sb, bno);
  char *buffer = bh->b_data; // les données de ce block
  ```

- Mise-à-jour l'offset du ficher : `*pos += copied_bytes; file→f_ops += copied_bytes;` et la fonction va retourne copied_bytes

La fonction ouichefs_write(struct file *file, char __user *data, size_t len, loff_t *pos) :

- La même manière que read(), remplacer copy_to_user() par copy_from_user()
- Avant la programme commence de l'écrire, il a besoin de vérifier l'existence du block :

  ```
  if (index->block[i] == 0) {
          // code pour allouer un nouveau block
          bno = get_free_block(OUICHEFS_SB(sb));
          index->blocks[iblock] = bno;
  ```

```
        // ...
}
```

- La mise-à-jour des informations du ficher : inode→i_size, inode→i_blocks, file→f_op et *pos

1.4 Modification of the data structure and implementation of an ioctl command

Pour réaliser notre but, nous avons besoin de parcourir les données stockées dans chaque block. OuicheFS utilise le champ index_block pour placer le numéro de block. Une boucle forte est implémentée pour examiner l'état d'un block.

```
for (int i = 0; i < 1024; i++){
        if (index->blocks[i] == 0) // arriver le dernier block
                break;
        // code pour vérifier l'état du block
}
```

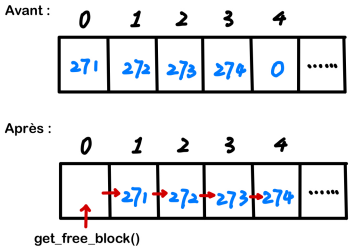
Le numéro de block est stocké en 4 bytes, et OuicheFS utilise 12bits du numéro de block pour présenter la taille de block, donc `int size = index->blocks[i] >> 20;`

- **the number of used blocks:** file→f_inode→i_blocks
- **the number of partially filled blocks:** Lorsqu'on parcours les blocks, s'il existe une block qui a une taille plus petit que 4096(OUICHEFS_BLOCK_SIZE), le compteur se augmente à 1
- **the number of bytes wasted due to internal fragmentation:** De même, `wasted_bytes += 4096 - size;`
- **the list of all used blocks with their number and effective size :** La format de la liste correspondant à [bno, effect_size]

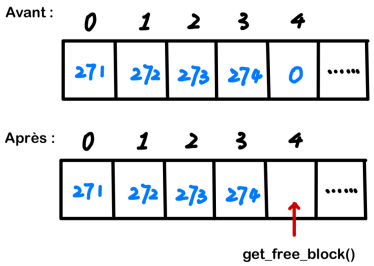
1.5 Implementation of the write function

Pour réaliser le "fast insert" write(), nous examinerons les opérations de la fonction d'écriture sur les blocs dans trois cas de figure :

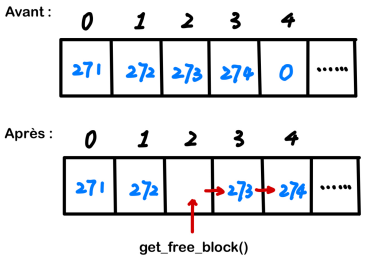
- Au début du ficher: Insère un nouveau bloc en tête de la liste des blocs

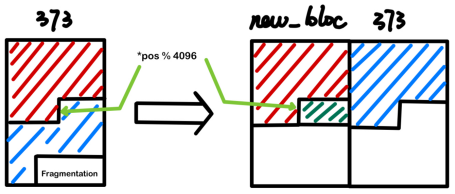


- A la fin du ficher: Insère un nouveau bloc à la fin de la liste directement



- Au milieu du ficher: S'il existe assez de place pour écrire (taille de fragmentation plus grande que len), la fonction insère le message directement à la contenu de ce bloc. Et s'il n'a pas, il faut ajouter un nouveau block.





Rouge : la contenu du ficher avant l'offset
Bleu : la contenu du ficher après l'offset
Vert : la message qu'on veut écrire

1.6 Modification of the read function

Le changement de lecture est plus simple, en bref, nous devrions sauter la lecture des fragmentations. Nous introduisons donc une variable globale, appelée "wasted".

Si le bloc en cours de lecture est partiellement utilisé et et cette lecture permet de lire tout le contenu restant de ce bloc :

```
// la taille de "wasted bytes"
wasted += 4096 - size;
// read() saute wasted bytes
*pos += copied_bytes + wasted
```

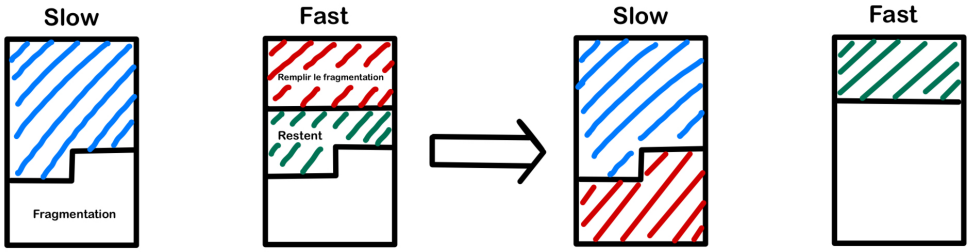
En même temps, on doit modifier la condition que lecture arrive à la fin : `if (*pos - wasted >= file->f_inode->i_size){` , sinon le read() perdra le contenu du dernier bloc.

1.7 Implementation of a defragmentation function

Pour réaliser cette fonctionalité, nous introduisons un fast-slow pointeur. Le slow pointeur corresponde l'index du premier bloc partiellement utilisé, et le fast pointeur corresponde l'index du bloc courant qui attend la défragmentation.

- si la taille de la fragmentation est plus grand que la taille de bloc de fast pointeur : Placez tout le texte stocké dans le bloc du pointeur fast dans le bloc du pointeur slow. Après l'insertion, l'index de fast pointeur se augmente à 1 (tous les données sont poussées)

- sinon, c-à-d qu'il n'a pas assez de place : Remplir complètement le fragment et placer les données restantes dans l'en-tête du bloc. Après l'insertion, l'index de slow pointeur a besoin de mettre à jour par la fonction `get_primier_wasted()`



- la condition de sortie : le nombre de caractère poussé (written) est plus grand que `file→inode→i_size`

Conclusion du projet :

Pour examiner la différence de la vitesse de l'execution, nous créons deux ficher : `/mnt/ouichefs/file1.txt` et `copy.txt`. Et le ficher source est `test.txt`

```
int file = open("test.txt", O_RDONLY);
copy_file(file, "/mnt/ouichefs/file1");
copy_file(file, "copy.txt");
```

Le fichier de test se compose de trois parties :

- La comparaison de la vitesse de l'insertion : `insert_file("Hello World", "/mnt/ouichefs/file1.txt", position);`
- La comparaison de la vitesse de lecture : `read_file("/mnt/ouichefs/file1.txt");`
- La comparaison de la vitesse de lecture dans Ouichefs avant et après la défragmentation : `ioctl(file1, DEFRAGEMENTATION, buf);  read_file("/mnt/ouichefs/file1.txt");`

La résultat de notre test est (Moyenne de 10 répétitions):

- read speed (normal, before defragmentation): 0.000134s
- read speed (normal, after defragmentation): 0.000135s
- read speed (ouichefs, before defragmentation): 0.000541s
- read speed (ouichefs, after defragmentation): 0.000307s
- write speed (normal, before defragmentation): 0.000185s
- write speed (ouichefs, after defragmentation): 0.000815s

Pour résumé, dans Ouichefs, l'écriture est moin rapide, ce qui peut être causer par l'action du dupliquer les données par la fonction memcpy(). Et la lecture est plus lent car il a besoin de lire plus

block comme habitue. En conséquence, nous avons besoin de modifier le ficher pour que les données du fichier soient continues, et cela peut diminuer le temps d’exécution de lecture.