310554034 蔡秉達

(1) FUSE 的目的 ：

Filesystem in Userspace，簡稱 FUSE，可以讓無特權使用者在無需編輯核心代碼的情況下，透過核心模組創建功能完備的文件系統。在此 final project 中，FUSE 被用來模擬 SSD，因此在沒有真實的硬體情況下，我們可以模擬 SSD 的讀寫以及 garbage collection。

(2) Flash Translation Layer (FTL)

FTL 是介於 SSD 前端和後端之間的一個轉換層，由於 SSD read modify write 的特性，也就是 SSD 無法直接覆寫，須經由抹除的時間 » 寫入的時間 » 讀取的時間，才不會讓資料壞掉，並且抹除和讀寫的單位不一樣，抹除的最小單位是一個 Block，讀寫的最小單位是一個 Page，以及藉由 L2P table 紀錄 logical block address 定應的 physical address，透過 FTL 讓 host interface 只需要對不同的 LBA 做存取，無須管 SSD 裡如 read modify write 等細節。

(3)修改地方 ：

1. **ssd_do_read**
2. **ftl_read**
3. **ssd_do_write**
4. **ftl_write**
5. **GC**

(4)實作方法

ssd_do_read :

由於已經算好page number以及要讀的page數量，故只需在323行做ftl_read將資料讀到buffer中。

```
303  static int ssd_do_read(char* buf, size_t size, off_t offset)
304  {
305      int tmp_lba, tmp_lba_range, rst ;
306      char* tmp_buf;
307
308      //off limit
309      if ((offset ) >= logic_size)
310      {
311          return 0;
312      }
313      if ( size > logic_size - offset)
314      {
315          //is valid data section
316          size = logic_size - offset;
317      }
318      tmp_lba = offset / 512;  // start from which page
319      tmp_lba_range = (offset + size - 1) / 512 - (tmp_lba) + 1;  // total page count
320      tmp_buf = calloc(tmp_lba_range * 512, sizeof(char));  // read data into this buffer
321
322      for (int i = 0; i < tmp_lba_range; i++) {  // read full pages
323          ftl_read(tmp_buf + i * 512, tmp_lba + i);
324      }
325
326      memcpy(buf, tmp_buf + offset % 512, size);
327
328
329      free(tmp_buf);
330      return size;
331  }
```

Ftl_read :

利用L2P table算出對應的pca，就能利用nand_read讀取資料。

```
228  static int ftl_read( char* buf, size_t lba)
229  {
230      nand_read(buf, L2P[lba]);
231  }
232
```

ssd_do_write:

由於SSD read modify write 的特性，需判斷頭尾page是否已有其他寫入的資料，有的話將其讀出一併寫入新的page內，其他page則直接寫入即可。

```
342  static int ssd_do_write(const char* buf, size_t size, off_t offset)
343  {
344      int tmp_lba, tmp_lba_range, process_size;
345      int idx, curr_size, remain_size, rst;
346      char* tmp_buf;
347
348      host_write_size += size;
349      if (ssd_expand(offset + size) != 0)
350      {
351          return -ENOMEM;
352      }
353
354      tmp_lba = offset / 512;
355      tmp_lba_range = (offset + size - 1) / 512 - (tmp_lba) + 1;
356
357      process_size = 0;
358      remain_size = size;
359      curr_size = 0;
360      size_t buf_idx = 0;
361      tmp_buf = calloc(512, sizeof(char));
362      for (idx = 0; idx < tmp_lba_range; ++idx) {
363          if (idx == 0 || idx == tmp_lba_range - 1) {
364              nand_read(tmp_buf, L2P[tmp_lba + idx]);
365              size_t begin = 0, end = 512;
366              if (idx == 0)
367                  begin = offset % 512;
368              if (idx == tmp_lba_range - 1)
369                  end = (offset + size) % 512;
370              if (end == 0)
371                  end = 512;
372              for (int i = begin; i < end; ++i)
373                  tmp_buf[i] = buf[buf_idx++];
374              ftl_write(tmp_buf, tmp_lba_range, tmp_lba + idx);
375          }
376          else {
377              ftl_write(buf + buf_idx, tmp_lba_range, tmp_lba + idx);
378              buf_idx += 512;
379          }
380      }
381      free(tmp_buf);
```

ftl_write:

利用 get_next_pca() 取得可以寫入的 page，取得 page 以後利用 nand_write 將資料寫入。若該 lba 之前已經被 map 到某個 pca 了，需要把舊的 pca 的狀態改為 stale。最後修改 P2L 及 L2P 更新 lab 和 pca 的 mapping，並將新的 pca 的狀態改為 valid

```
static int ftl_write(const char* buf, size_t lba_rnage, size_t lba)
{

    unsigned int pca = get_next_pca();
    if (free_block_number == 0){  //沒有下一個空的pca可拿
        GC();
        pca = get_next_pca();
    }
    nand_write(buf, pca);
    if (L2P[lba] != INVALID_PCA) {
        PCA_RULE old_pca;
        old_pca.pca = L2P[lba];
        unsigned int old_pca_idx = old_pca.fields.nand * PAGE_PER_BLOCK + old_pca.fields.lba;
        pca_status[old_pca_idx] = STALE_PCA;
    }
    PCA_RULE new_pca;
    new_pca.pca = pca;
    unsigned int new_pca_idx = new_pca.fields.nand * PAGE_PER_BLOCK + new_pca.fields.lba;
    P2L[new_pca_idx] = lba;
    L2P[lba] = pca;
    pca_status[new_pca_idx] = VALID_PCA;
}
```

GC :

在 ftl_write 裡，在做完 get_next_pca()拿到下一個空的 pca 後，會依據剩餘 free_block_number 的數量來判斷是否要做 GC，GC 使用暴力法，會檢查每個 block 裡的 page 是否為 valid page(該 block 中要有 stale page)，並搬到當前空的 block 中，也 earse 掉先前的的 block，若當前的 block 滿了就在換下一個空的 block，檢查完所有 block 後，也清完 invalid page 了，達到 garbage collection。

```
void GC() {
    for (int i = 1; i < PHYSICAL_NAND_NUM; ++i) {
        int idx = (curr_pca.fields.nand + i) % PHYSICAL_NAND_NUM;
        if (valid_count[idx] < PAGE_PER_BLOCK) {
            for (int j = idx; j < idx + PAGE_PER_BLOCK; ++j) {
                if(curr_pca.pca == FULL_PCA)
                    get_next_pca();
                if (pca_status[j] == VALID_PCA) {
                    char *buffer[512];
                    ftl_read(buffer, P2L[j]);
                    nand_write(buffer, curr_pca.pca);
                    //將L2P 、 P2L table 修改到對應的位子
                    L2P[P2L[j]] = curr_pca.pca;
                    P2L[curr_pca.fields.nand * PAGE_PER_BLOCK + curr_pca.fields.lba] = P2L[j];
                    P2L[j] = INVALID_LBA;

                    pca_status[j] = STALE_PCA;
                    pca_status[curr_pca.fields.nand * PAGE_PER_BLOCK + curr_pca.fields.lba] = VALID_PCA;
                    valid_count[curr_pca.fields.nand] += 1;
                    curr_pca.fields.lba += 1;
                }
            }
            valid_count[idx] = 0;
            nand_erase(idx);
        }
    }
}
```

(5)結果分析

在 test2 中，會寫入 122 個 page，當寫到 121 個 page 時，會沒有下一個空

的 pca 拿而做 GC，此時每個 block 會有 8 個 valid page 及 2 個 stale page，理想下他會清掉所有的 stale page，使 WA 來到 1。