SSD Firmware Implementation Project
- Lab. #6-

Sang-Phil Lim (lsfeel0204@gmail.com)

SKKU VLDB Lab.

2011-05-19

Lab. Time Schedule

Lab.	Title
#1	FTL Simulator Development Guide
#2	FTL Simulation Guide
#3	Project 1 Presentation
#4	Jasmine OpenSSD platform tutorial #1
#5	Jasmine OpenSSD platform tutorial #2
#6	**FTL Porting Guide**
#7	Firmware Debugging Guide
#8	SSD Performance Evaluation Guide
#9	Project 2 Presentation

FTL Porting Guide

Source file	Function	Description
./installer/installer.c	ftl_install_mapping_table	FTL 초기 메타데이터를 NAND 플래시에 기록하는 연산 수행. 펌웨어 설치 시 함께 호출됨
./ftl_[scheme]/ftl.c	ftl_open	FTL 초기화 과정 수행 - NAND 플래시로부터 메타데이터 로드 및 초기화 - VBLK #0 에 포맷 마크가 기록되어 있지 않을 경우, format 함수 호출
	format	VBLK #0 와 FTL 메타 영역을 제외한 나머지 블록들을 삭제 - 포맷 마크 기록
	ftl_read	사용자 데이터 읽기 처리
	ftl_write	사용자 데이터 쓰기 처리
	ftl_flush	SATA idle/standby time 에 주기적으로 메타데이터를 flush 하는 연산 수행

Introduction to Greedy FTL

- Page-level mapping FTL
- Features
  - Support simple garbage collection
    - Victim selection policy: *Greedy*
  - Support normal Power-Off Recovery
- Source files
  - `./ftl_greedy/ftl.h`
  - `./ftl_greedy/ftl.c`

Greedy FTL : FTL Metadata

- DRAM metadata
  - Page-level mapping table (`PAGE_MAP`)
    - For Logical-to-Physical page address mapping
  - Bad block bitmap table (`BAD_BLK_BMP`)
  - Block valid count information (`VCOUNT`)
    - For victim block selection
- SRAM metadata
  - Misc. information
    - Page index pointer of meta & user area
    - Remain free block count
    - Etc.

Greedy FTL : FTL Metadata

- Header file (**./ftl_greedy/ftl.h**)
  - **BAD_BLK_BMP, PAGE_MAP, VCOUNT**

```
...

////////////////////////////////
// DRAM segmentation
////////////////////////////////
...

#define BAD_BLK_BMP_ADDR     (TEMP_BUF_ADDR + TEMP_BUF_BYTES)          // bitmap of initial bad blocks
#define BAD_BLK_BMP_BYTES    (((NUM_VBLKS / 8) + DRAM_ECC_UNIT - 1) / DRAM_ECC_UNIT * DRAM_ECC_UNIT)

#define PAGE_MAP_ADDR        (BAD_BLK_BMP_ADDR + BAD_BLK_BMP_BYTES)    // page mapping table
#define PAGE_MAP_BYTES       ((NUM_LPAGES * sizeof(UINT32) + BYTES_PER_SECTOR - 1) / BYTES_PER_SECTOR
                             * BYTES_PER_SECTOR)

#define VCOUNT_ADDR          (PAGE_MAP_ADDR + PAGE_MAP_BYTES)         // valid page count per each vblk
#define VCOUNT_BYTES         ((NUM_BANKS * VBLKS_PER_BANK * sizeof(UINT32) + BYTES_PER_SECTOR - 1) /
                             BYTES_PER_SECTOR * BYTES_PER_SECTOR)
...
```

Greedy FTL : FTL Metadata

- Address alignment issue
  - Basically, memory address should be aligned 4 byte
  - In addition, in the DRAM case, 4 byte ECC parity is added to every 128 bytes of data

Greedy FTL: FTL Open

```
void ftl_open(void)
{
    // check DRAM footprint
    sanity_check();
    // build bitmap of bad blocks
    build_bad_blk_list();

    if (check_format_mark() == FALSE) {
        format();
    }
    // load FTL metadata
    else {
        // load FTL metadata in SRAM/DRAM from nand
        load_metadata();
    }
    // init FTL & SATA buffer pointer
    g_ftl_read_buf_id = 0;
    g_ftl_write_buf_id = 0;
}
```

Greedy FTL: Format

```
static void format(void)
{
    UINT32 bank, vblock, vcount_val;

    // initialize DRAM metadata
    mem_set_dram(PAGE_MAP_ADDR, NULL, PAGE_MAP_BYTES);
    mem_set_dram(VCOUNT_ADDR, NULL, VCOUNT_BYTES);

    // erase all blocks except vblock #0
    for (vblock = MISCBLK_VBN; vblock < VBLKS_PER_BANK; vblock++)
    {
        for (bank = 0; bank < NUM_BANKS; bank++)
        {
            vcount_val = VC_MAX;
            if (is_bad_block(bank, vblock) == FALSE)
            {
                nand_block_erase(bank, vblock);
                vcount_val = 0;
            }
            write_dram_32(VCOUNT_ADDR +
                          ((bank * VBLKS_PER_BANK) + vblock) * sizeof(UINT32),
                          vcount_val);
        }
    }
    ...
```

Greedy FTL: Format (contd.)

```
    ...
    // initialize SRAM metadata
    init_metadata_sram();

    // flush metadata to NAND
    logging_misc_metadata();
    logging_pmap_table();

    write_format_mark();
} // end of format()
```

SATA Controller: Handling IO Request

- Host send ATA command to SATA
- FIQ interrupt (./sata/sata_isr.c)

```
__irq void fiq_handler(void)
{
    ...
    if (masked_int_stat & CMD_RECV)
    {
        handle_got_cfis();
        intr_processed = CMD_RECV;
    }
    ...
}
static __inline void handle_got_cfis(void)
{
    ...
    else if (cmd_type & (CCL_FTL_H2D | CCL_FTL_D2H)) { // read or write req.
        // insert SATA event queue
    }
    ...
}
```

SATA Controller: SATA Event Queue

- Polling IO request
  - ./sata/sata_main.c

```
void Main(void)
{
    while (1){
        if (eventq_get_count()){
            CMD_T cmd;

            eventq_get(&cmd);

            if (cmd.cmd_type == READ)
                ftl_read(cmd.lba, cmd.sector_count);
            else
                ftl_write(cmd.lba, cmd.sector_count);
        }
        ...
```

SATA Controller: SATA Event Queue

- Polling IO request
  - **./sata/sata_main.c**

```
        ...
        else if (g_sata_context.slow_cmd.status == SLOW_CMD_STATUS_PENDING){
            void (*ata_function)(UINT32 lba, UINT32 sector_count);

            slow_cmd_t* slow_cmd = &g_sata_context.slow_cmd;
            slow_cmd->status = SLOW_CMD_STATUS_BUSY;

            ata_function = search_ata_function(slow_cmd->code); // sata.h
            ata_function(slow_cmd->lba, slow_cmd->sector_count); // sata_table.c

            slow_cmd->status = SLOW_CMD_STATUS_NONE;
        }
        else {
            // idle time operations
        }
    }
} // end of SATA main function
```

Greedy FTL : Read operation

```
void ftl_read(UINT32 const lba, UINT32 const num_sectors)
{
    lpn = lba / SECTORS_PER_PAGE;
    sect_offset  = lba % SECTORS_PER_PAGE;
    remain_sects = num_sectors;

    while (remain_sects != 0) {
        if ((sect_offset + remain_sects) < SECTORS_PER_PAGE) {
            num_sectors_to_read = remain_sects;
        }
        else {
            num_sectors_to_read = SECTORS_PER_PAGE - sect_offset;
        }
        bank = get_num_bank(lpn); // virtual page-level striping
        vpn  = get_vpn(lpn);      // address mapping
        ...
```

Greedy FTL : Read operation (contd.)

```
        ...
        // old data was already written
        if (vpn != NULL) {
            // send read request to nand flash
            nand_page_ptread_to_host(bank,
                                     vpn / PAGES_PER_BLK,
                                     vpn % PAGES_PER_BLK,
                                     sect_offset,
                                     num_sectors_to_read);
        }
        // The host is requesting to read a logical page that has
never been written to.
        else {
                // change Buffer Manager read limit pointer
                // change FTL SATA read pointer
        }
        sect_offset   = 0;
        remain_sects -= num_sectors_to_read;
        lpn++;
} // end of ftl_read function
```

Greedy FTL : Write Operation

```
void ftl_write(UINT32 const lba, UINT32 const num_sectors)
{
    UINT32 remain_sects, num_sectors_to_write;
    UINT32 lpn, sect_offset;

    lpn          = lba / SECTORS_PER_PAGE;
    sect_offset  = lba % SECTORS_PER_PAGE;
    remain_sects = num_sectors;

    while (remain_sects != 0) {
        if ((sect_offset + remain_sects) < SECTORS_PER_PAGE)
            num_sectors_to_write = remain_sects;
        else
            num_sectors_to_write = SECTORS_PER_PAGE - sect_offset;

        // single page write individually
        write_page(lpn, sect_offset, num_sectors_to_write);

        sect_offset   = 0;
        remain_sects -= num_sectors_to_write;
        lpn++;
    }
}
```

Greedy FTL : Write Operation (contd.)

```
static void write_page(UINT32 const lpn, UINT32 const sect_offset,
UINT32 const num_sectors)
{
    bank        = get_num_bank(lpn);
    page_offset = sect_offset;
    column_cnt  = num_sectors;
    old_vpn  = get_vpn(lpn);         // address mapping
    new_vpn  = assign_new_write_vpn(bank); // get free vpage

    // if old data already exist,
    if (old_vpn != NULL) {
        // read 'left hole sectors'  [left*][new data][right]
        if (page_offset != 0)
            nand_page_ptread(..., RETURN_ON_ISSUE);
        // read `right hole sectors'  [left][new data][right*]
        if ((page_offset + column_cnt) < SECTORS_PER_PAGE)
            nand_page_ptread(..., RETURN_ON_ISSUE);
        // invalid old page (decrease vcount)
        set_vcount(bank, vblock, get_vcount(bank, vblock) - 1);
    }
    ...
```

Greedy FTL : Write Operation (contd.)

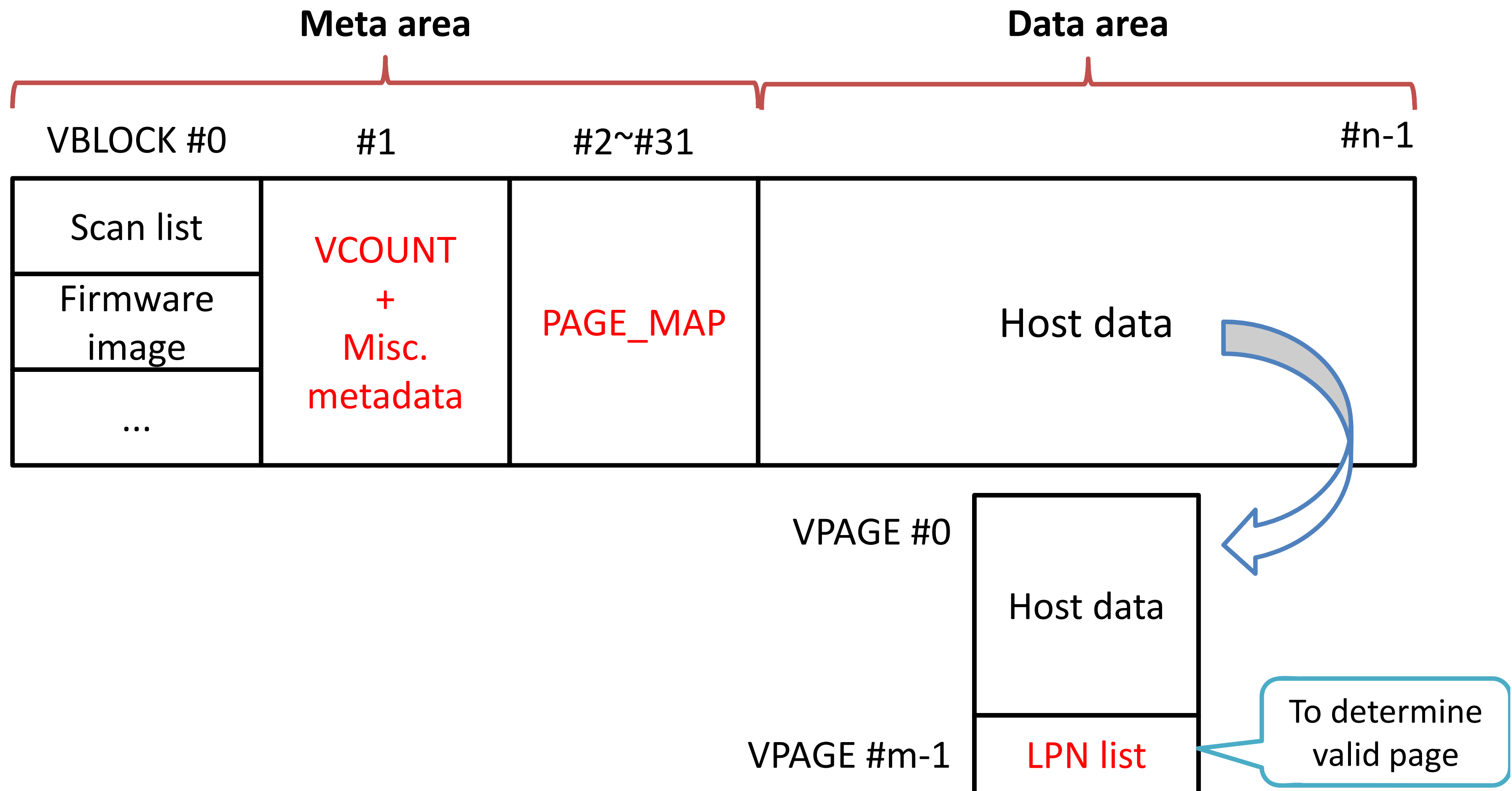
```
    ...
    vblock    = new_vpn / PAGES_PER_BLK;
    page_num = new_vpn % PAGES_PER_BLK;

    // write new data (RETURN_ON_ISSUE)
    nand_page_ptprogram_from_host(bank,
                                  vblock,
                                  page_num,
                                  page_offset,
                                  column_cnt);
    /* update metadata */
    set_lpn(bank, page_num, lpn); // maintain lpn list for GC
    set_vpn(lpn, new_vpn);        // address mapping table
    // increase block valid page count
    set_vcount(bank, vblock, get_vcount(bank, vblock) + 1);

} // end of write_page function (caller: ftl_write)
```

Flash Command

- To send I/O command to NAND flash
  ① Programming FCP manually
    - See tutorial FTL code (`./ftl_tutorial/ftl.c`)

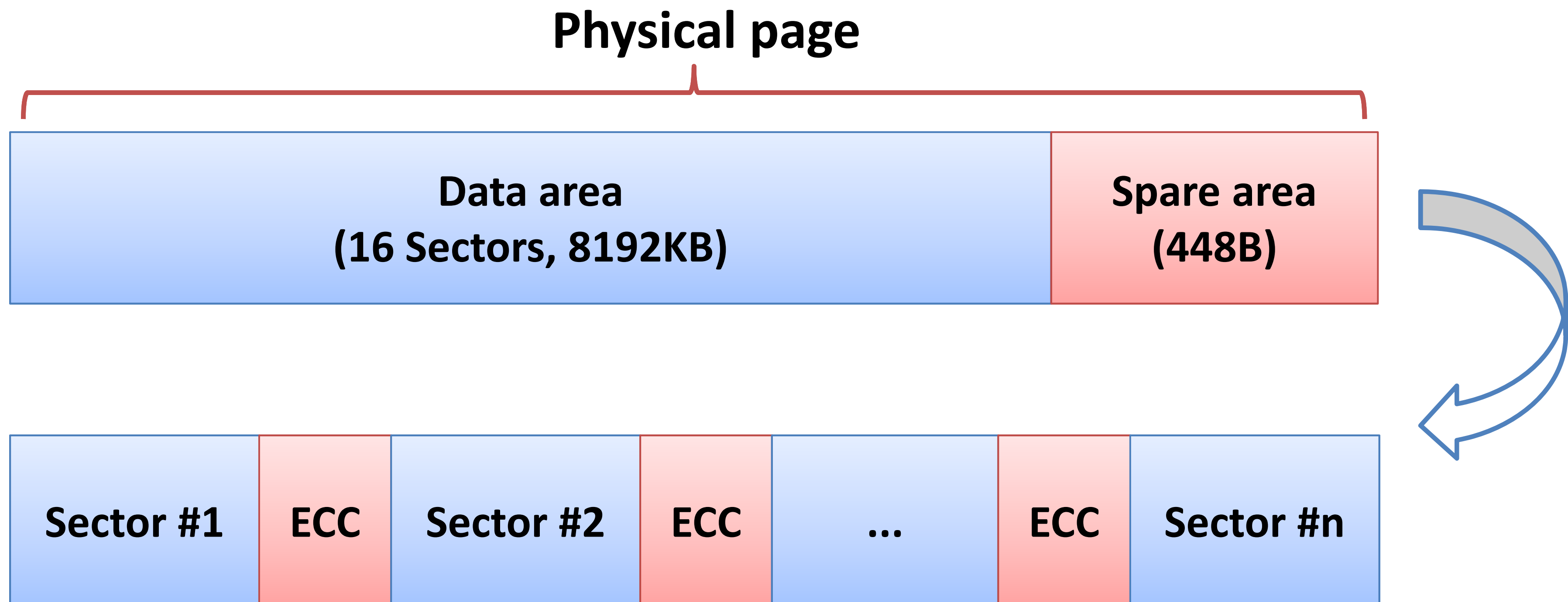
  ② Use LLD interface (`./target_spw/flash_wrapper.c`)
    - Easiest way to implement an FTL
    - If you want to make a high performance FTL, the first option is recommended

Greedy FTL : NAND Configuration



Restriction of Accessing NAND

- No support for accessing 'spare area'



Greedy FTL : Garbage Collection

- If free blocks run out, a victim block is chosen based on the 'greedy policy'
  - i.e., select the block containing min. valid pages

```
static UINT32 get_vt_vblock(UINT32 const bank)
{
    UINT32 vblock;

    // search the block which has mininum valid pages
    vblock = mem_search_min_max(VCOUNT_ADDR + (bank * VBLKS_PER_BANK * sizeof(UINT32)),
                                sizeof(UINT32),
                                VBLKS_PER_BANK,
                                MU_CMD_SEARCH_MIN_DRAM);
    return vblock;
}
```

Greedy FTL : Garbage Collection

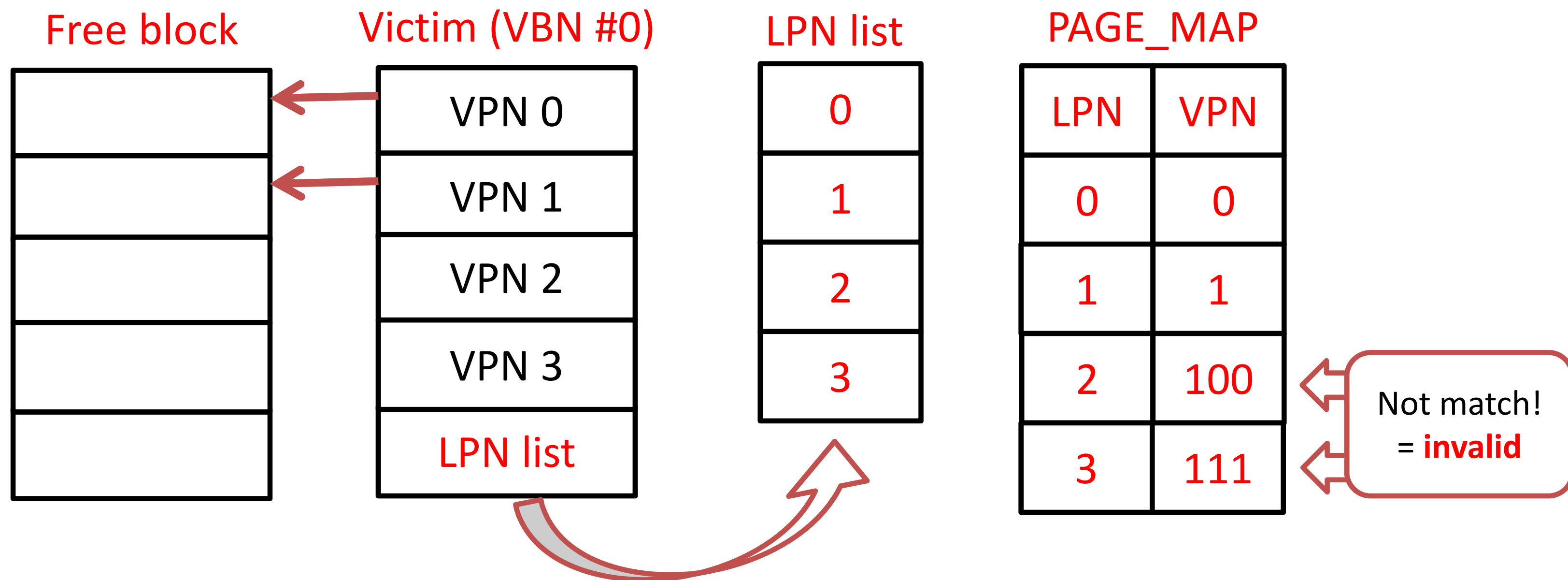
```
static void garbage_collection(UINT32 const bank)
{
    ...
    // 1. load p2l list from last page offset of victim block
    nand_page_ptread(bank, vt_vblock, PAGES_PER_BLK - 1, 0, 1,
                     FTL_BUF_ADDR, RETURN_WHEN_DONE);
    mem_copy(g_lpn_list_of_cur_vblock[bank],
             FTL_BUF_ADDR, sizeof(UINT32) * PAGES_PER_BLK);

    // 2. copy-back all valid pages to free space
    for (src_page = 0; src_page < (PAGES_PER_BLK - 1); src_page++) {
        src_lpn = get_lpn(bank, src_page);

        // determine whether the page is valid or not
        if (get_vpn(src_lpn) !=
            ((vt_vblock * PAGES_PER_BLK) + src_page)) {
            // invalid page
            set_lpn(bank, src_page, INVALID);
            continue;
        }
        // if the page is valid, do copy-back op. to free space
        nand_page_copyback(bank, vt_vblock, src_page, free_vpn / PAGES_PER_BLK,
                                                      free_vpn % PAGES_PER_BLK);
        /* update metadata */
        set_vpn(src_lpn, free_vpn);
        set_lpn(bank, (free_vpn % PAGES_PER_BLK), src_lpn);
        set_lpn(bank, src_page, INVALID);
        free_vpn++;
    }
    ...
```

Greedy FTL : Garbage Collection

```
    ...
    // 3. erase victim block
    nand_block_erase(bank, vt_vblock);

    // 4. update metadata
    set_vcount(bank, vt_vblock, VC_MAX);
    set_vcount(bank, gc_vblock, vcount);
    set_new_write_vpn(bank, free_vpn); // set a free page for new write
    set_gc_vblock(bank, vt_vblock); // next free block (reserve for GC)
    dec_full_blk_cnt(bank); // decrease full block count
}
```

Greedy FTL : Garbage Collection

- Internal operation



Greedy FTL : POR(Power-Off Recovery)

- Metadata logging
  - Flush all metadata instantly (by `ftl_flush`)
  - @ SATA ready/idle/standby time
  - Fixed metadata area

VBLK #0	#1	#2~#31	
Scan list Firmware image ...	VCOUNT + Misc. metadata	PAGE_MAP	User space

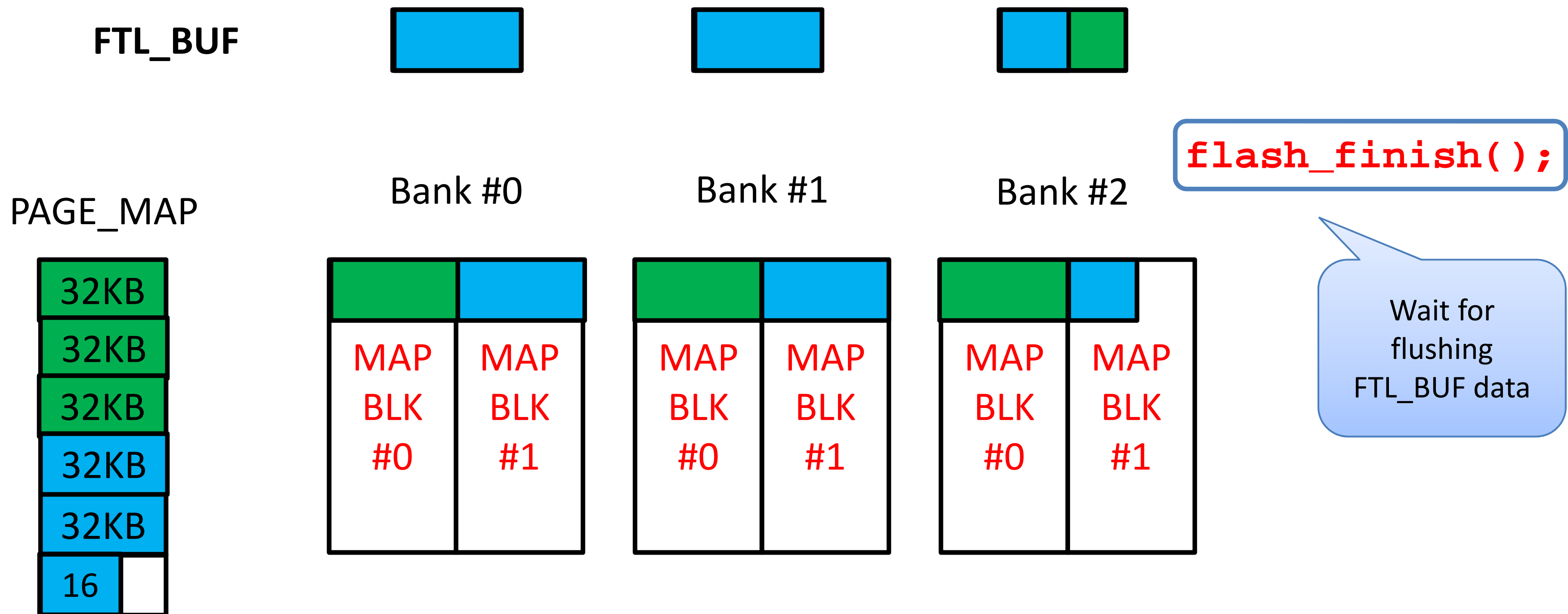
Greedy FTL : Logging PAGE_MAP

```
static void logging_pmap_table(void)
{
    UINT32 base_addr = PAGE_MAP_ADDR;
    UINT32 pmap_bytes;
    UINT32 mapblk_vpn;
    UINT32 bank;
    UINT32 pmap_boundary = PAGE_MAP_ADDR + PAGE_MAP_BYTES;

    for (UINT32 mapblk_lbn = 0; mapblk_lbn < MAPBLKS_PER_BANK; mapblk_lbn++) {
        flash_finish();

        for (bank = 0; bank < NUM_BANKS; bank++) {
            // set next map block vpn
            inc_mapblk_vpn(bank, mapblk_lbn);

            mapblk_vpn = get_mapblk_vpn(bank, mapblk_lbn);

            // logging update page mapping info. into map block
            // if there is no free page, then erase old map block first.
            if ((mapblk_vpn % PAGES_PER_BLK) == 0) {
                // erase full map block
                nand_block_erase(bank, (mapblk_vpn - 1) / PAGES_PER_BLK);

                // next vpn of mapblk is offset #0
                set_mapblk_vpn(bank, mapblk_lbn,
                               ((mapblk_vpn - 1) / PAGES_PER_BLK) * PAGES_PER_BLK);
            }
            ...
```

Greedy FTL : Logging PAGE_MAP (contd.)

```
            ...

            if (base_addr > pmap_boundary)  {
                break;
            }
            else if ((base_addr + BYTES_PER_PAGE) > pmap_boundary) {
                pmap_bytes = PAGE_MAP_BYTES % BYTES_PER_PAGE;
            }
            else {
                pmap_bytes = BYTES_PER_PAGE;
            }
            mem_copy(FTL_BUF_ADDR + (bank * BYTES_PER_PAGE), base_addr, pmap_bytes);

            // logging page mapping info to NAND
            nand_page_program(bank,
                              get_mapblk_vpn(bank, mapblk_lbn) / PAGES_PER_BLK,
                              get_mapblk_vpn(bank, mapblk_lbn) % PAGES_PER_BLK,
                              FTL_BUF_ADDR + (bank * BYTES_PER_PAGE));
            base_addr += pmap_bytes;
        }
    }
}
```

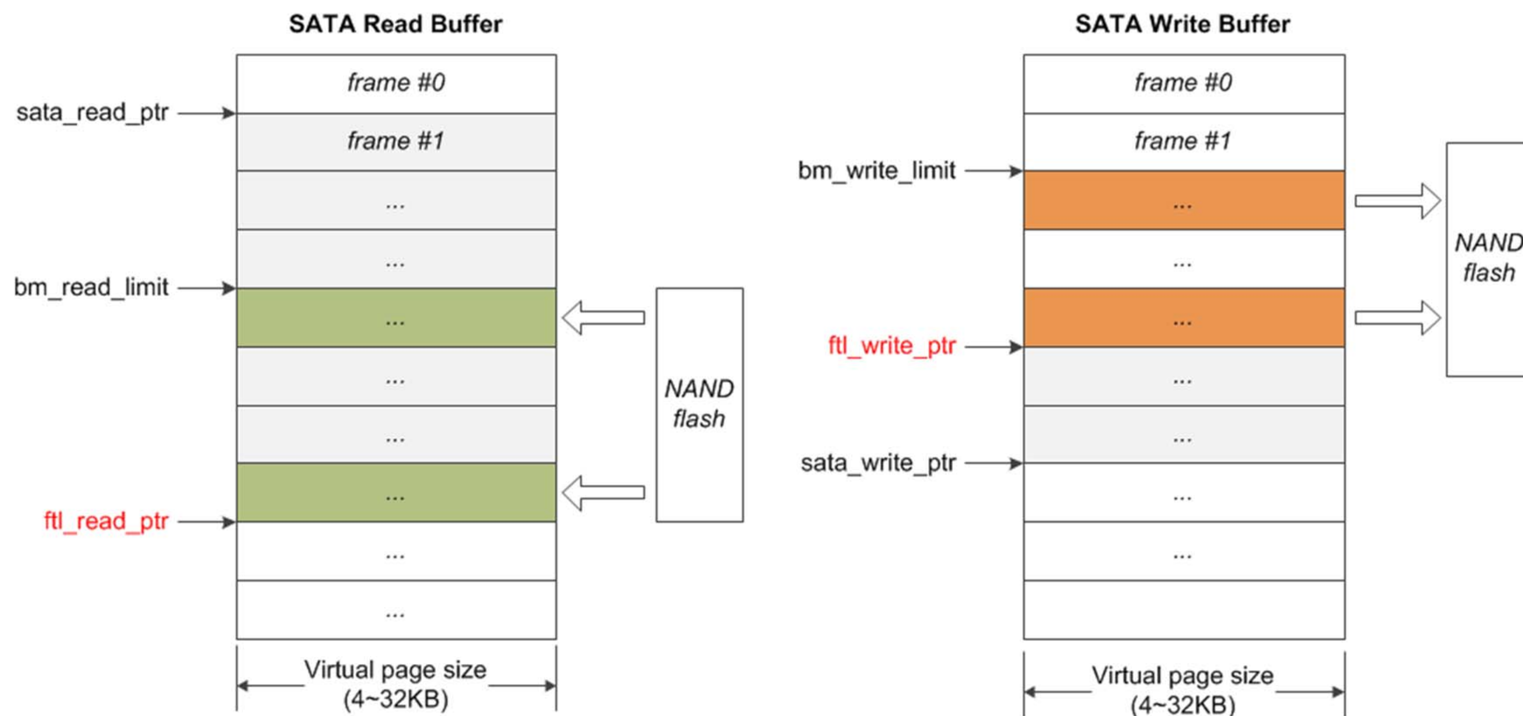
Greedy FTL : Logging PAGE_MAP



Supplement of Previous Lab.

c.f.) SATA Host Buffer

- Basically, SATA & BM buffer pointer are controlled by H/W
- However, in SATA read buffer, the flow control between SATA and FTL is necessary by firmware



c.f.) SATA Host Buffer

```
void nand_page_read_to_host(UINT32 const bank, UINT32 const vblock, UINT32 const
page_num)
{
    g_ftl_read_buf_id = (g_ftl_read_buf_id + 1) % NUM_RD_BUFFERS;

#if OPTION_FTL_TEST == 0
    while (1)
    {
        UINT32 sata_id = GETREG(SATA_RBUF_PTR);

        if (g_ftl_read_buf_id != sata_id)
            break;
    }
#endif

    flash_issue_cmd(bank, RETURN_ON_ISSUE);
}
```

Any Questions?