SW중심대학

LAB 01 : MyFTL

October 1, 2025
Jeyeon Lee
Dept. of Software
Dankook University
jeyeonlee@dankook.ac.kr

(본 교재는 2025년 과학기술정보통신부 및 정보통신기획평가원의 'SW중심대학사업' 지원을 받아 제작 되었습니다.)

Content

과제 목표

- Greedy FTL 시뮬레이터 구현
  - 요청 처리 및 응답
    - 요청된 명령어, 주소, 길이를 확인할 수 있다
    - 명령어를 구분하여 알맞게 처리할 수 있다
  - 매핑 일관성 유지
    - LPN(Logical Page Number)과 PPN(Physical Page Number)을 매핑할 수 있다
    - 상황에 맞추어 매핑 테이블을 관리(map, remap, unmap)할 수 있다
  - 내부 자원 관리
    - 요청에 맞춰 새로운 페이지를 할당할 수 있다
    - GC(Garbage Cleaning)를 수행하여 가용 자원을 확보할 수 있다
    - Greedy GC 알고리즘을 구현할 수 있다

과제 준비

- 리눅스 환경에서 수행하는 것을 권장
  - 요구 라이브러리
    - Ubuntu 22.04
    - gcc 11.4
    - make 4.3
- 과제 구성
  - 깃 복제 `git clone https://github.com/DKU-EmbeddedSystem-Lab/2025_DKU_SSP.git`
  - 구성

```
2025_DKU_SSP/
├ lab1/
├ lab2/
└ README.md
```

```
lab1/
├ workload/
|  ├ sequential.csv
|  └ random.csv
├ Makefile
├ test.h
├ test.c
├ myftl.h
└ myftl.c
```

워크로드 파일

빌드 및 테스트를 위한 파일들

구현 대상 파일

구현 후 실행

```
make
./myftl
make clean
```

```
lab1/
├ sequential_mapping_table.csv
└ random_mapping_table.csv
```

매핑 덤프 파일 생성됨

과제 준비

- 구현이 필요한 대상 함수 확인

```c
/* TODO 구현 필요한 함수(필요시 자유롭게 추가) */
int ftl_io(struct ssd *ssd, struct workload_entry cmd);
int ftl_read(struct ssd *ssd, uint64_t lpn, int page_count);
int ftl_write(struct ssd *ssd, uint64_t lpn, int page_count);
int ftl_gc(struct ssd *ssd);
```

myftl.h

- 요청 확인

```c
int ftl_io(struct ssd *ssd, struct workload_entry cmd) {
    /* TODO
     * 요청을 확인하고 알맞은 처리 함수 호출
     */
    printf("ftl_read 구현 필요\n");
    return -1;  // 구현되지 않음
}
```

myftl.c

- 쓰기

```c
int ftl_write(struct ssd *ssd, uint64_t lpn, int page_count) {
    for (int i = 0; i < page_count; i++) {
        ssd->total_writes++;
        /* TODO
         * LPN에 새로운 PPA 할당
         * 이전 매핑이 있으면 invalid로 표시
         * 새로운 페이지 할당 및 매핑 테이블 업데이트
         */
        printf("ftl_write 구현 필요\n");
        return -1;  // 구현되지 않음

        // GC 필요성 체크
        if (ssd->sm.free_sb_cnt <= ssd->sm.gc_thres_sbs) {
            ftl_gc(ssd);
        }
    }
    return 0;
}
```

myftl.c

- 읽기

```c
int ftl_read(struct ssd *ssd, uint64_t lpn, int page_count) {
    for (int i = 0; i < page_count; i++) {
        ssd->total_reads++;
        /* TODO
         * LPN을 PPA로 변환하여 읽기 연산 수행
         * 매핑 테이블에서 LPN에 해당하는 PPA를 찾아 읽기
         */
        printf("ftl_read 구현 필요\n");
        return -1;  // 구현되지 않음
    }
    return 0;
}
```

myftl.c

- GC

```c
int ftl_gc(struct ssd *ssd) {
    ssd->total_gc_cnt++;
    /* TODO
     * victim 슈퍼블록 선택
     * valid 데이터를 새로운 위치로 복사
     * (복사한 페이지 수만큼 ssd->total_gc_pages 증가)
     * 블록 삭제 및 슈퍼블록 상태 업데이트
     */

    printf("ftl_gc 구현 필요\n");
    return -1;  // 구현되지 않음
}
```

myftl.c

(기본 함수 형태 제외 자유롭게 추가/변경 가능)

MyFTL 구조

- MyFTL 내부 자원 및 관리 구조
  - channel, die, block, page
  - maptbl: LPN-PPN 매핑 테이블
  - rmap: GC 수행 시 maptbl 업데이트를 위한 PPN-LPN 역-매핑 테이블
  - wp: 다음 할당될 페이지의 위치 유지
  - sm: 슈퍼 블록(superblock) 관리

```c
struct ssd {
    struct channel ch[NCHS];
    struct ppa maptbl[TOTAL_USER_PGS];
    uint64_t rmap[TOTAL_PGS];
    struct write_pointer wp;
    struct sb_mgmt sm;

    // 통계
    uint64_t total_reads;
    uint64_t total_writes;
    uint64_t total_gc_cnt;
    uint64_t total_gc_pages;
};
```
myftl.h

```c
struct channel {
    struct die die[DIES_PER_CH];
};
```
myftl.h

```c
struct die {
    struct block blk[BLKS_PER_DIE];
};
```
myftl.h

```c
struct block {
    struct page pg[PGS_PER_BLK];
};
```
myftl.h

```c
struct page {
    int status;
};
```
myftl.h

```c
struct write_pointer {
    // 다음 쓰기 수행할 위치
    int ch;
    int lun;
    int pg;
    int blk;
    int pl;
};
```
myftl.h

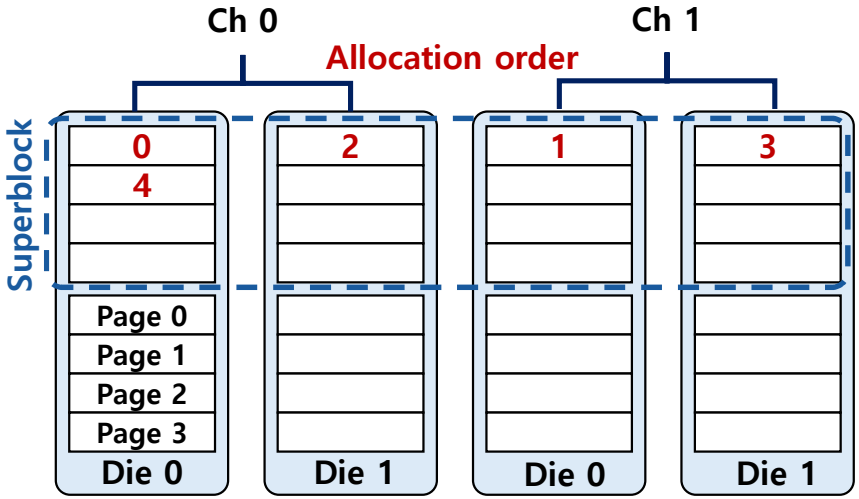
```c
struct ppa {
    union {
        struct {
            uint64_t blk : 16;
            uint64_t pg  : 16;
            uint64_t die : 16;
            uint64_t ch  : 16;
        } g;
        uint64_t ppa;
    };
};
```
myftl.h

```c
#define PGS_PER_BLK     128
#define BLKS_PER_DIE    64
#define DIES_PER_CH     8
#define NCHS            8
```
<내부 구성 설정 값>

MyFTL 구조

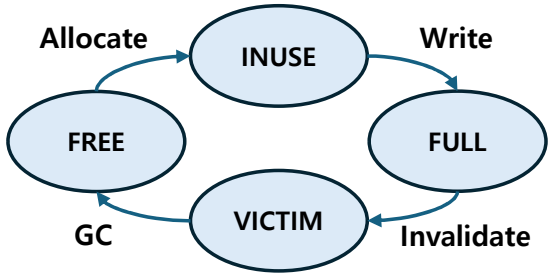
- MyFTL은 내부 자원을 슈퍼 블록 단위로 관리
  - 슈퍼 블록: 모든 다이들의 블록 집합
    - MyFTL에선 단순히 같은 번호의 블록들로 구성

```c
struct superblock {
    int blk_id;
    int status;
    int ipc;  // invalid page count
    int vpc;  // valid page count
};
```
myftl.h



  - 내부 페이지들은 채널-다이-페이지 순으로 할당
  - 슈퍼블록의 상태에 따라 관리
    - SB_FREE: 사용되지 않은 상태
    - SB_INUSE: 현재 페이지 할당 대상 (wp가 이 슈퍼블록 내부 페이지를 가리키고 있음)
    - SB_FULL: 모든 슈퍼블록의 페이지들이 사용된 상태
    - SB_VICTIM: 슈퍼블록 내부에 유효하지 않은 페이지가 존재할 때

```c
SB_FREE = 0,
SB_VICTIM = 1,
SB_FULL = 2,
SB_INUSE = 3,
```
<정의된 SB 상태>



MyFTL 동작 과정

- ftl_io()
  - I/O 요청 확인
    - 요청의 명령어, 논리 주소, 크기 등 추출
    - 요청 구조체는 test.h에서 확인 가능
  - ftl_read/write() 호출
    - 명령어 역시 test.h에 정의됨
    - 명령어에 알맞은 함수 호출 및 논리 주소, 길이 전달
  - 완료 응답
    - MyFTL은 별도의 완료 응답 과정 없음

```c
struct workload_entry {
    int operation;      // 0: READ, 1: WRITE
    uint64_t lpn;       // Logical Page Number
    int page_count;     // 페이지 수
};
```
test.h

```c
typedef enum {
    NVM_IO_READ = 0,
    NVM_IO_WRITE = 1
} operation_type;
```
test.h

```c
int ftl_io(struct ssd *ssd, struct workload_entry cmd) {
    op, start_lba, len = get_from(cmd);
    if (is_write(op)) {
        return ftl_read(start_lba, len);
    } else if (is_read(op)) {
        return ftl_write(start_lba, len);
    }
    return CMD_INVALID;
}
```
pseudo_myftl.c

MyFTL 동작 과정

- ftl_read()
  - 시작 주소로부터 요청된 페이지 수 만큼 읽기 수행
  - 접근할 주소 유효성 확인
    - 주소가 용량이 허용하는 범위를 넘어서는지 확인
  - 논리 주소-물리 주소 변환
    - 매핑 테이블 참조
    - 매핑되지 않은 논리주소는 0 데이터 반환
    - 매핑된 논리주소는 물리주소에 읽은 후 반환
    - MyFTL은 실제로 데이터를 읽거나 반환하지 않음

```c
int ftl_read(struct ssd *ssd, uint64_t lpn, int page_count) {
    current_lpn = start_lpn;
    while (page_count) {
        if (!valid_lpn(current_lpn))
            return LPN_INVALID;

        ppa = lpn_to_ppa(current_lpn);
        if (!mapped_ppa(ppa))
            read_zeros();

        current_lpn++;
    }
    return 0;
}
```

pseudo_myftl.c

MyFTL 동작 과정

- ftl_write()
  - 이전에 쓰였던 데이터의 존재 여부 확인
    - 있다면 해당 페이지를 유효하지 않은 상태로 업데이트
    - 역-매핑 테이블 역시 업데이트
  - 새로운 페이지 할당
    - 현재 쓰기 포인터가 가리키는 페이지 할당
    - 쓰기 포인터 업데이트
    - 현재 슈퍼블록이 가득 찼다면 새 슈퍼블록 할당
  - 매핑 테이블 업데이트
    - 새로운 페이지로 역-/매핑 테이블 업데이트
  - 가용 슈퍼블록이 부족해진다면 GC 수행

```c
int ftl_write(struct ssd *ssd, uint64_t start_lpn, int page_count) {
    current_lpn = start_lpn;
    while (page_count) {
        if (!valid_lpn(current_lpn))
            return LPN_INVALID;

        old_ppa = lpn_to_ppa(current_lpn);
        if (mapped_ppa(old_ppa)) {
            mark(old_ppa, INVALID);
            update_rmap(old_ppa);
        }

        new_ppa = get_next_page();
        update_maptbl(current_lpn);
        mark(new_ppa, VALID);
        update_rmap(new_ppa);

        if (should_gc(ssd))
            ftl_gc(ssd);

        current_lpn++;
    }
    return 0;
}
```

pseudo_myftl.c

MyFTL 동작 과정

- ftl_gc()
  - 가장 적은 유효 페이지를 가진 슈퍼블록을 GC 대상으로 선정
  - 슈퍼 블록 내부 유효 페이지 이주
    - 모든 페이지 순회하며 유효성 확인
    - 유효한 페이지는 쓰기 포인터 위치로 이주
  - 블록 삭제
    - 내부 유효한 페이지가 없어진 블록은 삭제
    - 블록 상태 업데이트
  - 슈퍼블록 상태 업데이트

```c
int ftl_gc(struct ssd *ssd) {
    victim = select_victim();
    if (!victim)
        return NO_VICTIM

    for (block = victim->blocks) {
        for (page = block->pages) {
            if (!valid(page))
                continue;

            gc_write(page);
            mark(page, INVALID);
        }
        erase(block);
        mark(block, FREE);
    }

    mark(victim, FREE);

    return 0;
}
```

pseudo_myftl.c

과제 제출

- 구현 완료 후 수행 시 매핑 테이블 덤프 파일 생성됨
  - 수행 방법은 p.4 참고

- 소스 파일과 덤프 파일 .tar 형식으로 묶은 후 보고서와 함께 제출
  - tar –cvf 학번_lab1.tar lab1/

- 제출 형식
  - 소스코드 압축 파일: 학번_lab1.tar
  - 보고서: 학번.pdf

- 파이팅

```
lab1/
├ workload/
|  ├ random.csv
|  └ sequential.csv
├ Makefile
├ test.h
├ test.c
├ myftl.h
├ myftl.c
├ sequential_mapping_table.csv
└ random_mapping_table.csv
```

<압축 파일에 포함돼야할 파일 목록>

Acknowledgement

- 본 교재는 **2025**년도 과학기술정보통신부 및 정보통신기획평가원의 **'SW중심대학사업'** 지원을 받아 제작 되었습니다.