Pusan National University
Computer Science and Engineering
Technical Report 2025-09

ZNS SSD를 지원하는 POSIX 파일시스템 개발

202025126 김민준

202055548 박천휘

지도교수: 안성용

목 차

1. 요약

기존의 block SSD는 호스트의 워크로드(workload) 특성을 알지 못하기 때문에, 공간과 시간 자원을 비효율적으로 활용하는 문제가 존재한다. 이러한 문제를 극복하기 위해 새로운 NAND flash 기반 저장장치인 ZNS SSD가 등장했지만, 이를 완전히 지원하는 파일시스템은 여전히 부족하다. 대표적인 ZNS SSD 지원 파일시스템으로는 f2fs와 zonefs가 있으나, f2fs는 순수 ZNS SSD 환경에서 단독으로 구동할 수 없고, zonefs는 사용성이 떨어진다는 한계가 있다.

본 과제에서는 이러한 문제를 해결하기 위해 순수 ZNS SSD를 직접 지원하면서도 POSIX API를 제공하여 높은 사용성을 갖춘 파일시스템을 설계 및 구현하였다.

2. 서론

2.1. ZNS SSD의 배경 설명

2.1.1. 기존 SSD의 한계

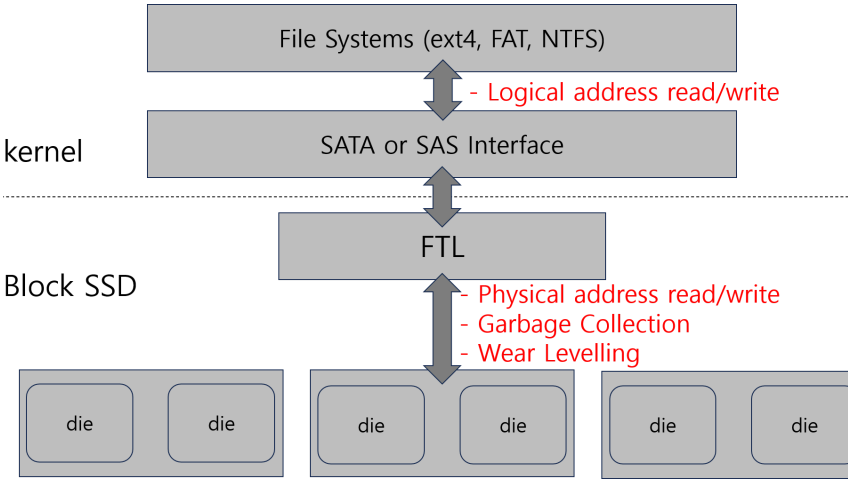
SSD(Solid State Drive)는 NAND flash 소자로 구성된 보조기억장치다. SSD는 HDD(Hard Disk Drive)에 비해 빠른 읽기 속도를 가지고, 외부의 물리적 충격에 대해 더 강한 내구성을 지녔다. 그러나 SSD는 다음과 같은 한계도 지닌다.

1. 쓰기가 수행된 SSD의 셀은 지우기 연산을 거쳐야만 새로운 값이 쓰여질 수 있다.
2. SSD에서 지우기 연산의 단위인 블록은 쓰기 연산의 단위보다 크다.
3. SSD의 각 셀은 쓰기 연산과 지우기 연산에 의해 마모된다.

따라서 SSD에 쓰여진 데이터는 제자리에서 덮어쓰는 방식으로 업데이트하는 것보다 새 지점에 최신 데이터 값을 쓰는 방식으로 업데이트하는 것이 효율적이다. 이러한 업데이트 방식으로 인해 SSD에는 유효한 데이터와 유효하지 않은 데이터가 혼재되어 있어, 추후 새로운 데이터를 쓸 수 있도록 빈 블록을 확보하는 garbage collection이 필요하다. 그리고 특정 블록에 garbage collection으로 인한 마모가 집중되는 것을 예방하기 위한 wear leveling도 필요하다.

SSD의 이러한 한계를 숨기고, 기존의 HDD와 유사한 동작을 보이기 위해 Block SSD는 그림 1과 같이 하드웨어나 펌웨어 수준에서 FTL(Flash Translation Layer)이 주소 매핑, garbage collection, wear leveling을 수행하도록 한다.

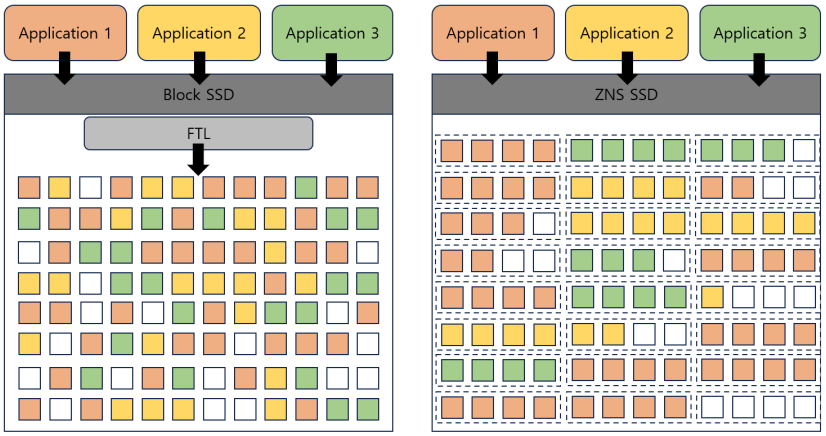
그러나 FTL은 SSD의 자체 프로세서와 메모리를 필요로하며, 효율적인 garbage collection을 수행하기 위해 여유 공간을 남겨두는 오버프로비저닝(Over-Provisioning)이 필요해 SSD의 유효 용량을 감소시킨다. 그리고 하드웨어 수준에서 알 수 있는 워크로드에 대한 컨텍스트는 제한적이기에 효율적인 garbage collection과 wear leveling을 구현하는 데 한계가 있다. 비효율적인 garbage collection은 사용자의 쓰기 연산 요청을 병목 시켜 SSD의 쓰기 성능을 감소시키고, 비효율적인 wear leveling은 특정 소자에 마모를 집중시켜 SSD의 수명이 빠르게 단축시키는 문제점으로 이어진다.



<그림 1: Block SSD에서 FTL의 역할>

2.1.2. ZNS SSD 설명

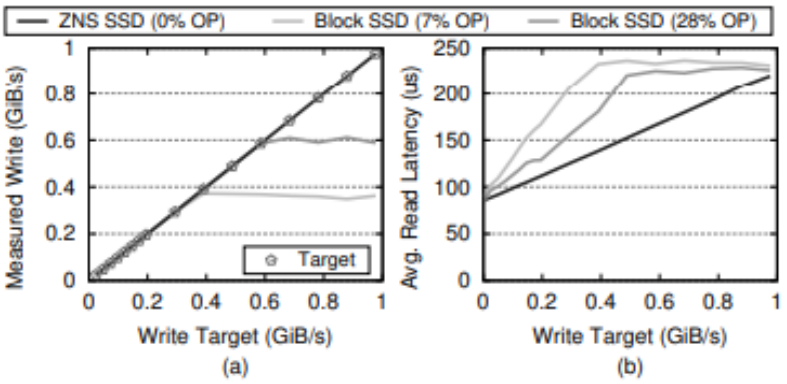
Block SSD의 문제점을 극복하고자, 호스트에게 기존 FTL의 책임을 이전하는 ZNS(Zoned Name Space) SSD가 등장했다. ZNS SSD는 그림 2처럼 여러 개의 블록을 모아 zone이라는 단위를 구성하고, zone 안에서는 순차적 쓰기만을 허용한다. 호스트는 특정 zone에 순차적으로 데이터를 쓰거나, zone 전체를 리셋하며 직접 SSD를 관리하게 된다.



<그림 2: 기존 Block SSD와 ZNS SSD의 데이터 저장 방식 차이>

ZNS SSD는 호스트의 CPU와 메모리를 활용하여 기존 FTL의 기능을 수행하기에 SSD 자체의 프로세서와 DRAM 비용을 절감할 수 있다. 그리고 워크로드의 컨텍스트를 알고 있는 호스트가 직접 쓰기 연산을 관리하므로 garbage collection의 효율을 높일 수 있고, 각 zone에 리셋 횟수를 조절하여 wear leveling을 보다 효과적으로 수행할 수 있다.

ZNS SSD는 그림 3처럼 기존 Block SSD와 달리 선형에 가까운 쓰기 연산 처리량을 보이며, 이는 garbage collection으로 인한 쓰기 증폭이 적게 발생하기 때문이다. ZNS SSD는 오버프로비저닝이 필요하지 않으면서도 Block SSD에 비해 1.7~2.7배 더 높은 쓰기 처리량을 달성할 수 있다.[1]



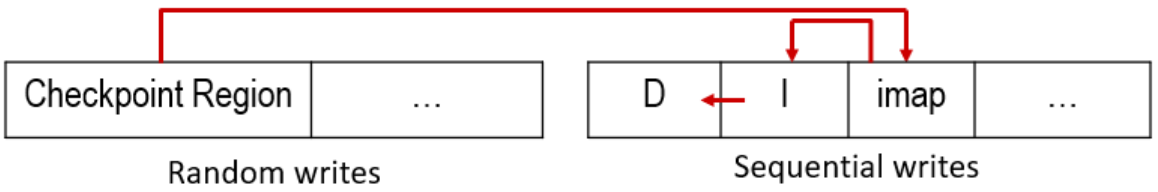
<그림 3: ZNS SSD와 Block SSD의 쓰기 처리량, 읽기 지연 차이>

2.2. ZNS SSD를 지원하는 파일시스템

ZNS SSD를 비롯한 순차적 쓰기 제약을 가지는 ZBD(Zoned Block Device)는, 제자리 업데이트 방식을 사용하는 파일시스템과는 부적합하다. 따라서 순차적 쓰기를 사용하도록 설계된 f2fs(Flash Friendly File System), 혹은 각 zone을 파일로 다루는 파일 시스템인 zonefs가 적합하다. 그러나 이들 파일시스템은 여러 한계를 가지고 있으며, 그중 f2fs의 한계를 먼저 살펴본다.

2.2.1. f2fs의 한계

f2fs(Flash Friendly File System)는 로그 구조(Log-Structured) 파일 시스템으로, 데이터를 기존의 자리에서 업데이트하지 않고, 최신 데이터를 새 위치에 순차적으로 써가며 업데이트한다. 순차적인 쓰기 방식을 사용하므로 아이노드와 같은 메타데이터를 비롯한 데이터들의 현재 유효한 위치를 가리키는 포인터가 필요하며, 데이터의 유효한 주소가 바뀌면 이 데이터의 위치를 가리키는 포인터도 업데이트해야 한다는 문제점이 있다. f2fs를 비롯한 로그 구조 파일 시스템은 그림 4와 같이 데이터들의 유효한 위치를 직접 가리키지 않도록 imap이라는 간접층을 둠으로써, 데이터가 새로운 위치에 업데이트되더라도 매핑 테이블에서만 업데이트가 일어나도록 하였다. 다만 흩어져 있는 imap의 위치를 가리키는 체크포인트가 고정된 위치에 쓰여야 하므로, f2fs는 순차적 쓰기만이 가능한 sequential zone으로 이뤄진 순수한 ZNS SSD는 지원하지 않는다는 한계가 있다.[2]

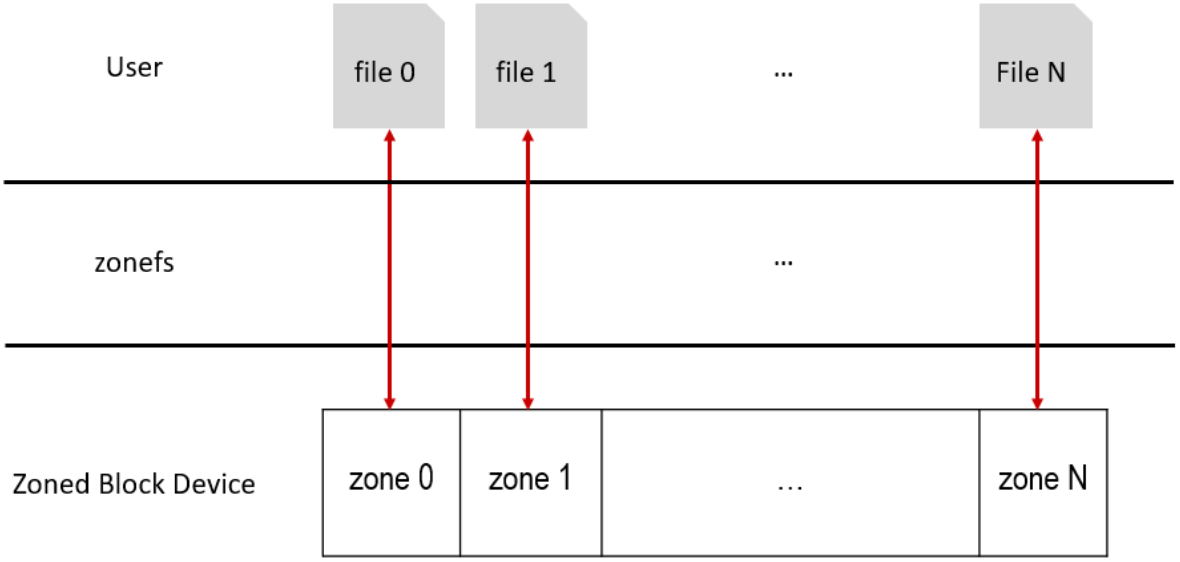


<그림 4: f2fs 구조 개요>

2.2.2. zonefs의 한계

ZBD의 순차 쓰기 제약 특성을 활용하거나 연구하기 위한 특수 목적의 파일 시스템으로 zonefs가 개발되었고, 현재 리눅스 커널에 모듈로 탑재되어 있다.

zonefs는 ZNS SSD의 각 zone을 하나의 일반 파일로 매핑하며, 사용자에게 순차 쓰기 제약을 그대로 노출한다. 그렇기에 zonefs의 기능은 일반적인 파일 시스템이라기보다는 그림 5와 같이 raw block device 접근 라이브러리에 가깝다.



<그림 5: zonefs에서 zone과 file의 매핑 관계>

zonefs는 ZNS SSD와 같은 ZBD에 대한 접근을 단순화하고, 애플리케이션이 직접 장치의 쓰기 포인터를 다루지 않아도 되도록 추상화 계층을 제공한다. 그러나 이러한 설계는 다음과 같은 한계를 가진다.

(1) POSIX 기능 미제공

zonefs는 최소한의 파일시스템 정보를 담은 슈퍼블록만을 생성한다. 파일에 대한 메타데이터는 존재하지 않으며, 따라서 POSIX의 기능 대부분이 구현되어 있지 않다. 사용자가 디렉토리를 만들 수 없으며, zone이 아닌 새로운 파일을 생성할 수도 없다. 이는 zonefs가 ZBD를 위한 범용 파일 시스템으로 활용되기 어려움을 의미한다.

(2) zone 할당 정책의 부재

zonefs는 각 파일을 zone에 일대일 매핑한다. 이는 파일시스템의 구현을 단순화하는 장점이 있지만, zone의 공간 활용도나 쓰기 병렬성을 고려한 동적 할당이나 최적화 정책이 부재하다는 문제를 가진다. 파일을 사용할 때 zone 단위의 고정된 매핑 구조로 인해 공간 단편화(fragmentation)가 발생하거나, 효율적인 zone 재사용이 어려워진다. 이와 같은 단순 매핑 모델은 대규모 데이터 워크로드나 고성능 요구 환경에서 성능 병목의 원인이 될 수 있다.

(3) 사용자 부담 증가

zonefs는 효율적인 garbage collection과 wear leveling의 책임을 사용자에게 이전한다. 그러나 사용자가 직접 유효하지 않은 데이터를 식별하고, 마모 정도가 낮은 zone을 파악하도록 하는 기존 zonefs의 방식은 사용자 부담을 증가시킨다.

2.3. 과제 목표

2.3.1. POSIX API 제공

현재 zonefs는 아이노드와 같은 메타데이터 구현의 부족으로, POSIX 기반 리눅스 명령어(ls, cat, echo, mkdir 등)가 지원되지 않는다. 이를 지원하도록 리눅스 커널의 zonefs를 확장 구현하는 것을 목표로 한다.

2.3.2. 효율적인 garbage collection 기능 구현

사용자는 POSIX 파일시스템 API를 사용하며, ZNS SSD는 단순히 append 연산만 수행한다. 이러한 방식은 불가피하게 garbage를 발생시키므로, 중간 계층인 파일시스템에 garbage collection을 구현하여 불필요한 데이터를 줄이고 저장 공간 활용도를 높이고자 한다. 결과적으로 사용자는 POSIX 인터페이스만 이용하면 되므로 장치의 공간 관리에 대한 부담이 줄어들 것이다.

2.4. 과제 수행 개요

Linux 커널 버전 6.8의 zonefs 모듈 소스코드를 수정 및 추가하여 본 과제의 파일시스템을 구현했음. VFS에서 호출하는 zonefs의 operation 중, POSIX 기능을 제공하지 않는 operation들이 이를 제공할 수 있도록 함수들을 수정하는 방식으로 이루어졌음. 성능 테스트는 fio를 이용하였음.

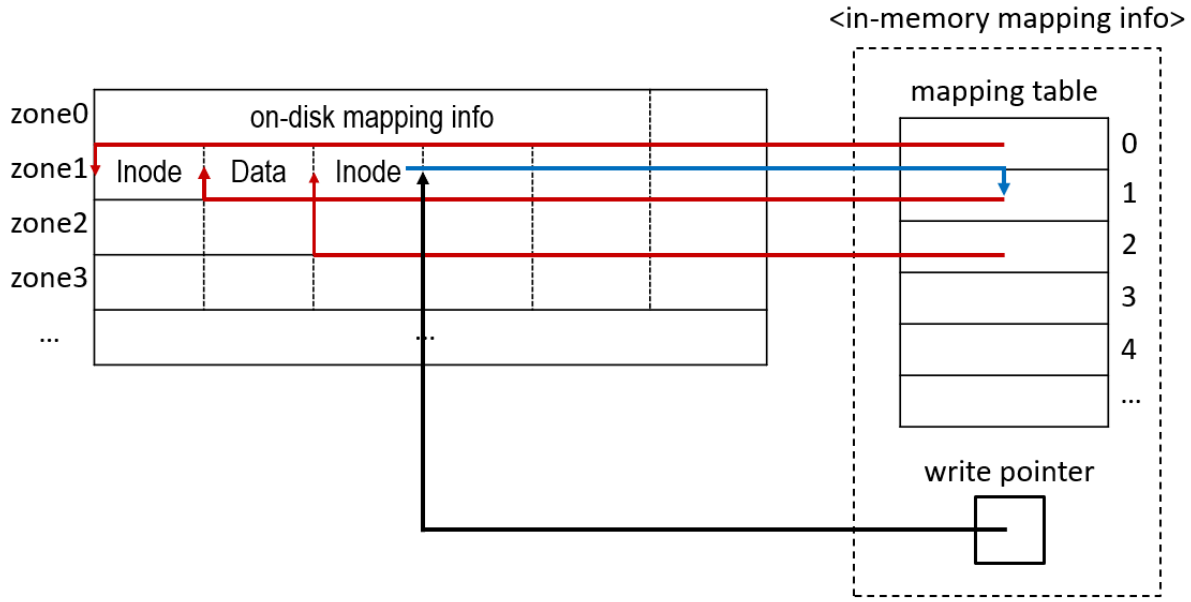
2.5. 과제 결과

본 파일시스템은 POSIX 파일시스템 API를 지원하여 계층적인 디렉토리 구조 생성 및 파일의 읽기-쓰기가 가능하다. 또한, 장치의 공간 효율성을 향상시키기 위하여 garbage collection 기능을 추가적으로 구현하였다. 성능 측정 결과, 파일 읽기 속도는 평균 3 MiB/s, 쓰기 속도는 평균 17.5 MiB/s를 보였으며, garbage collection은 유효 데이터량 대비 평균 7.16 MiB/s의 처리 속도를 기록하였다.

3. 본론

3.1. 설계

3.1.1. 설계 개요



<그림 6: 파일시스템 설계 구조도>

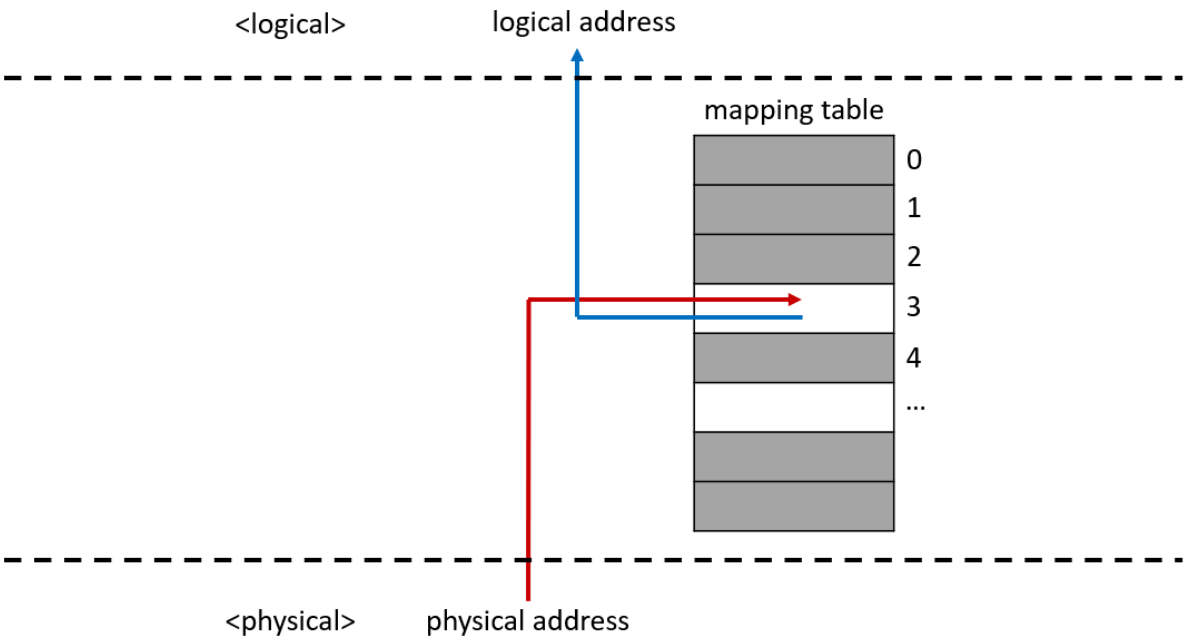
본 파일시스템의 구조는 그림 6과 같다. zone 1번부터 아이노드와 데이터 블록이 현재 write pointer 위치에 순차적으로 혼재된 채 저장된다. 유효한 아이노드와 데이터 블록은 고유한 논리 주소를 갖고, 이들의 물리적 위치는 메모리의 매핑 테이블에서 관리된다. 아이노드는 자신이 나타내는 파일의 데이터 블록 논리 주소를 보유한다. 따라서 파일 접근은 아이노드가 참조하는 데이터 블럭의 논리 주소를 통해 매핑 테이블로부터 물리적 위치를 확인하는 방식으로 이루어진다. 이 매핑 테이블은 in-memory mapping info에 포함되어 메모리에 존재하며, zone 0번에 주기적으로 저장되고 마운트 시 로드된다.

3.1.2. 매핑 테이블

매핑 테이블은 아이노드나 데이터 블록의 논리 주소와 물리 주소의 매핑 관계를 담고 있다. ZNS SSD에서는 순차적 쓰기만이 허용되므로, 데이터의 업데이트마다 데이터의 물리적 위치가 매번 바뀐다.

이렇게 새로운 쓰기가 발생하면 해당 데이터 블록을 가리키는 아이노드나 간접 데이터 블록의 포인터 값도 바뀌어야 하므로, 대상으로부터 포인터를 거슬러가며 새로운 쓰기가 연쇄적으로 일어나게 된다. 이러한 추가적인 쓰기 소요를 막기 위해, 논리 주소와 물리 주소 사이의 매핑을 이용한 간접층인 매핑 테이블을 사용하도록 한다.

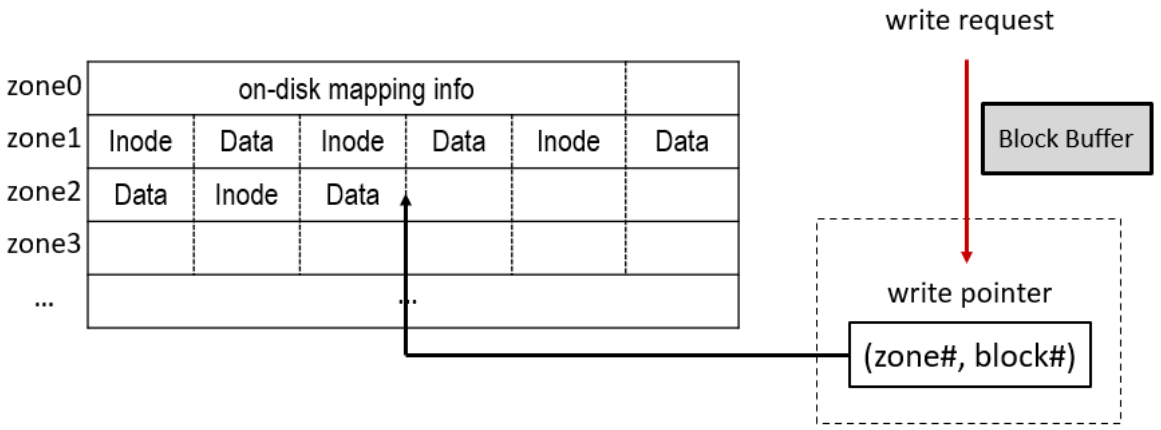
새로 쓰인 데이터 블록이나 아이노드의 물리 주소를 매핑 테이블에 등록하는 경우는 그림 7과 같이 물리적 계층으로부터 온 물리 주소를 매핑 테이블의 빈 항목에 추가하고 해당하는 논리 주소를 논리 계층에 반환한다. 그 후에 매핑 테이블을 통해 특정 논리 주소의 물리 주소를 검색하는 경우엔 앞선 과정을 반대로 수행한다.



<그림 7: 새로운 데이터 블록이나 아이노드를 매핑테이블에 등록하는 과정>

하나의 데이터 블록의 크기가 4KB, 하나의 물리 주소가 4-Byte이므로 총 저장 용량의 0.1%에 해당하는 크기의 매핑 테이블 공간이 필요하다. 매핑 테이블 전체를 메모리에 충분히 올릴 수 있기에, 매핑 테이블을 메모리에 올린 채로 사용하여 파일 시스템의 속도를 높이고, 이를 zone 0번에 주기적으로 저장하고 부팅 시 로드 하도록 하여.

3.1.3. 쓰기 포인터

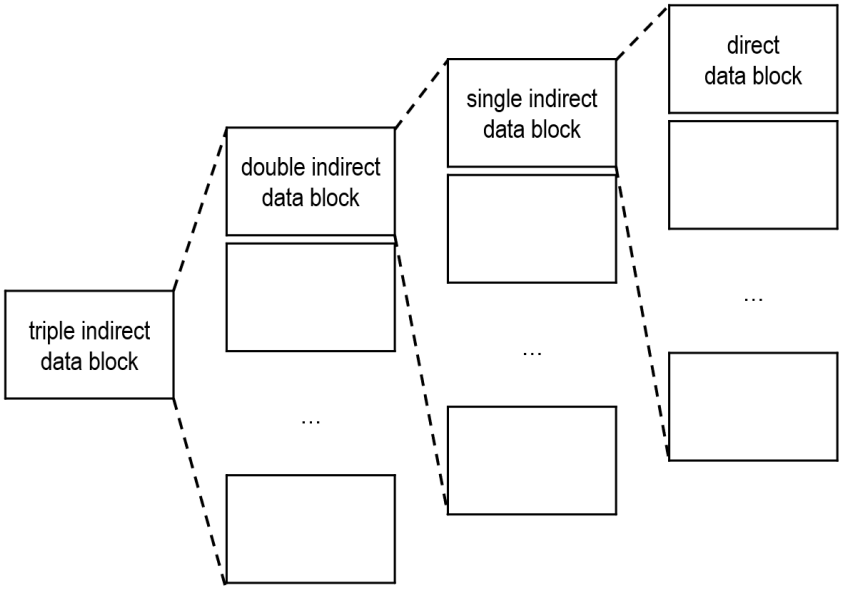


<그림 8: 쓰기 요청 과정에서 쓰기 포인터의 동작>

쓰기 포인터는 ZNS SSD 장치에서 다음에 쓰여질 물리 주소를 가리키는 역할을 하며, 이는 zone 번호와 zone 내 블록 번호로 표현된다. 그림 8은 장치 쓰기 요청 과정에서 쓰기 포인터의 동작을 나타낸다. 쓰기 요청이 발생하면, 파일시스템은 쓰고자 하는 내용을 담은 Block Buffer와 함께 장치에 쓰기를 요청한다. 이때 장치 쓰기 계층은 쓰기 포인터가 가리키는 물리 주소에 Block Buffer 데이터를 기록한다. 이러한 구조는 상위 계층이 물리적 쓰기 위치를 직접 관리할 필요 없이 쓰기 요청만 수행할 수 있도록 추상화를 제공한다. 또한, 쓰기 포인터는 ZNS SSD의 순차적 쓰기 제약을 반영하여 동작한다. 즉, 쓰기가 완료될 때마다 쓰기 포인터는 자동으로 다음 물리 주소로 이동하며, 이를 통해 장치 내부의 순차 쓰기 규칙이 자연스럽게 유지된다.

3.1.4. 데이터 블록

데이터 블록은 사용자가 파일에 기록한 payload 데이터가 담기는 direct block과, 이들을 가리키는 논리 주소를 기록하여 더 큰 용량을 간접적으로 가진 indirect block으로 나뉜다. 그리고 indirect block들은 얼마나 간접적으로 direct block을 가리키는지에 따라 single indirect block부터 triple indirect block까지 총 3종류로 나뉜다. 그림 9은 triple indirect block이 다른 indirect block을 거쳐 direct block을 간접적으로 가리키는 구조를 나타낸 그림이다.



<그림 9: triple indirect block이 가리키는 데이터 블록들>

데이터 블록을 구분하기 위해 밑의 그림 10과같이, 데이터 블록은 간접 깊이를 나타내는 매직 넘버를 포함한다. direct 데이터 블록은 'D' 'A' 'T' '0'을 매직 넘버로 가지고, single indirect 데이터 블록은 'D' 'A' 'T' '1', double indirect 데이터 블록은 'D' 'A' 'T' '2', triple indirect 데이터 블록은 'D' 'A' 'T' '3'을 매직 넘버로 가진다.

그리고 매직 넘버 외에 해당 데이터 블록에 할당된 논리 주소도 포함한다. 이 덕분에 파일 시스템이 데이터 블록을 임의 위치로 옮겨야 하는 garbage collection 등의 작업을 수행할 때, 데이터 블록 위치의 변화를 매핑 테이블에 쉽게 반영할 수 있다.

data block

magic number ex : "DAT0"	logical number ex : 16	file data	..			

<그림 10: 논리 주소 16번을 할당 받은 direct block의 예시>

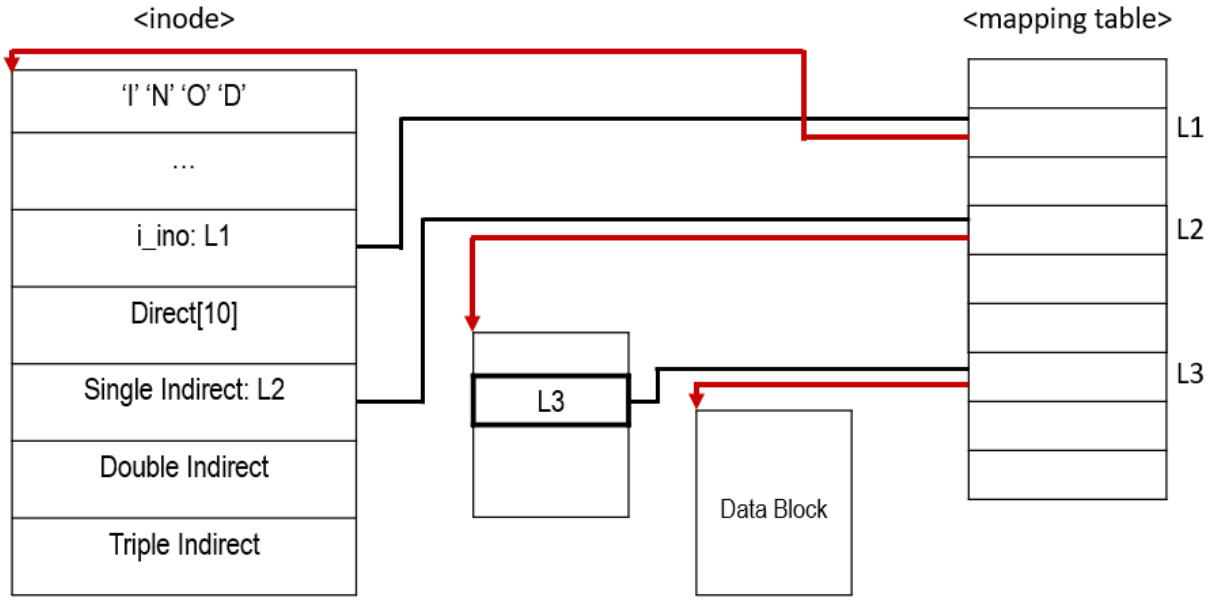
3.1.5. 아이노드

아이노드는 파일의 메타데이터를 저장하며, 본 파일시스템에서는 데이터 블록과 아이노드가 ZNS SSD에 혼재되어 저장된다. 이를 구분하기 위해 각 블록에는 magic 필드가 존재하며, 아이노드 블록의 경우 'I', 'N', 'O', 'D' 값이 기록된다.

아이노드 번호 i_ino는 새로운 파일 생성 시에 할당되는 논리적 블록 주소로 초기화된다. 즉, 아이노드 번호는 곧 해당 아이노드 블록의 논리적 블록 번호로 정의된다.

파일의 데이터 블록 관리는 멀티 레벨 인덱스 구조를 기반으로 한다. 인덱싱은 논리적 블록 주소를 통해 수행된다. 그림 11은 single indirect block을 이용하여 데이터 블록에 접근하는 과정을 보여준다. 구체적으로, 아이노드의 single indirect block 엔트리에 저장된 논리적 블록 주소를 참조하여 해당 블록을 읽어온 뒤, 그 안에 포함된 direct block의 논리적 블록 주소를 이용해 실제 데이터 블록에 접근한다. double indirect block과 triple indirect block 또한 동일한 원리로 동작한다.

논리적 블록 주소의 크기를 4바이트, 블록 크기를 4KB로 설정하였기 때문에, 본 파일시스템이 지원하는 단일 파일의 최대 크기는 약 4TB이다.

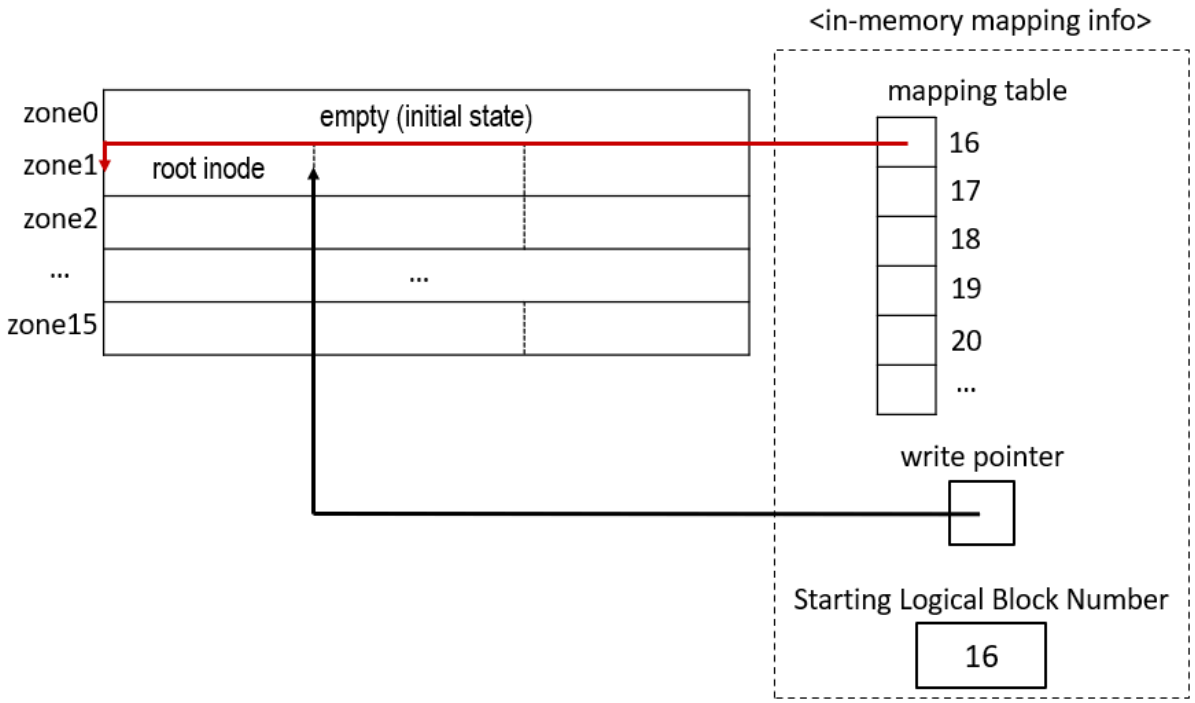


<그림 11: 아이노드 구조와 멀티 레벨 인덱스 블록을 통한 데이터 블록 접근 과정>

3.2. 구현

3.2.1. 초기화

파일시스템이 마운트 되면 초기화 과정이 수행된다. 먼저 ZNS SSD의 zone 0에서 on-disk mapping info를 읽어와 in-memory mapping info를 구성한다. 만약 파일시스템이 최초로 마운트 되는 경우라면, in-memory mapping info를 새롭게 초기화한 뒤, 그림 12와 같이 루트 디렉토리를 생성한다. 루트 디렉토리는 항상 시작 논리 블록 번호를 부여받으며, 이후 파일 경로 탐색 시 이 고정된 루트 디렉토리의 i_ino 값을 사용한다. 여기서 주목할 점은, 본 파일시스템에서 시작 논리 블록 번호를 ZNS SSD의 zone 개수로 설정한다는 것이다. 이는 마운트 과정에서 각 zone마다 아이노드가 하나씩 부여되기 때문이다. 본 파일시스템은 아이노드 번호를 곧 논리 블록 번호로 사용하므로, 아이노드 번호의 중복을 방지하기 위해 시작 논리 블록 번호를 기준으로 아이노드 번호(논리 블록 번호)를 계산한다.



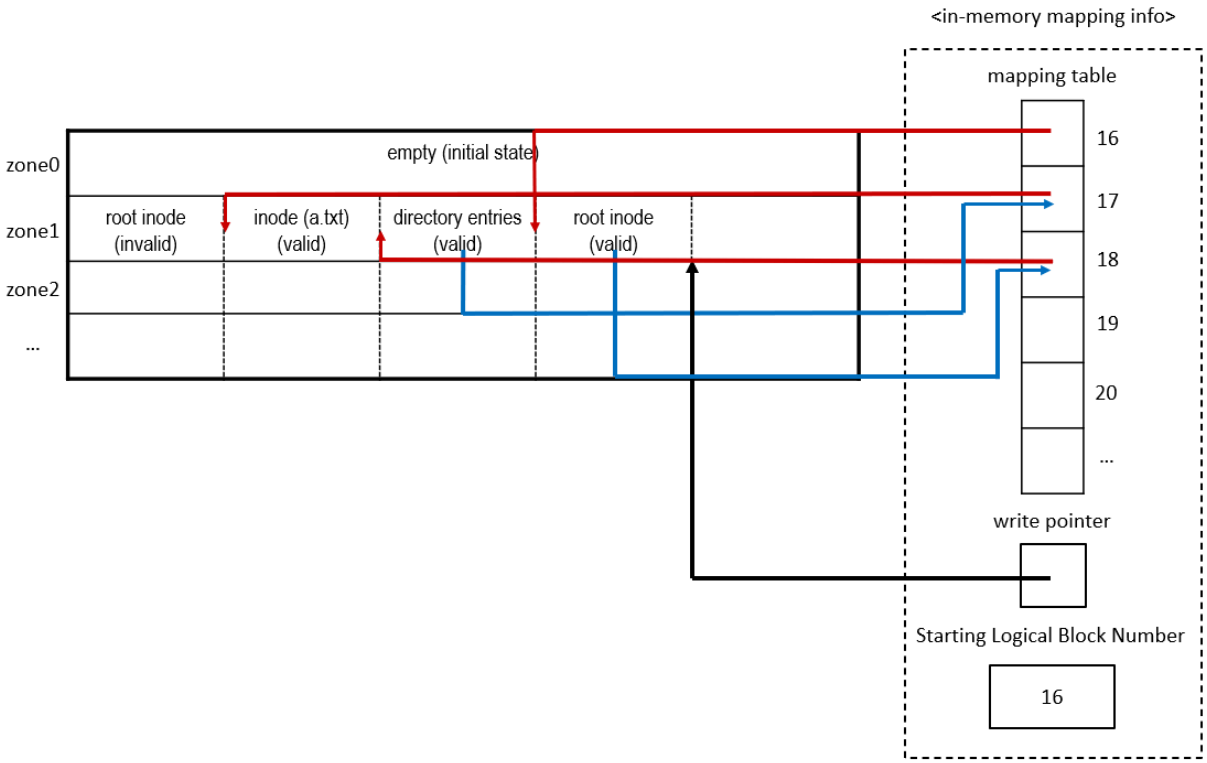
<그림 12: 파일시스템 초기화 직후 상태: zone 개수가 16개인 경우>

3.2.2. 파일/디렉토리 생성

파일 생성 과정에서는 VFS inode와 on-disk inode가 함께 생성된다. 이때 비어 있는 논리 블록 주소를 할당받아 아이노드의 i_ino에 저장하고, 새로 생성된 파일은 부모 디렉토리의 디렉토리 엔트리에 추가된다.

본 파일시스템은 ZNS SSD의 순차 쓰기 제약을 반영한다. 따라서 기존 블록의 내용에 변경이 발생하면, 해당 블록을 invalid 처리한 뒤 새로운 위치에 변경된 내용을 기록한다. 그 결과, 그림 13과 같이 새로운 파일이 생성되고 관련 데이터가 ZNS SSD에 순차적으로 저장된다.

디렉토리 생성 과정은 파일 생성 과정과 유사하게 동작한다.



<그림 13: a.txt 파일 생성 후 상태>

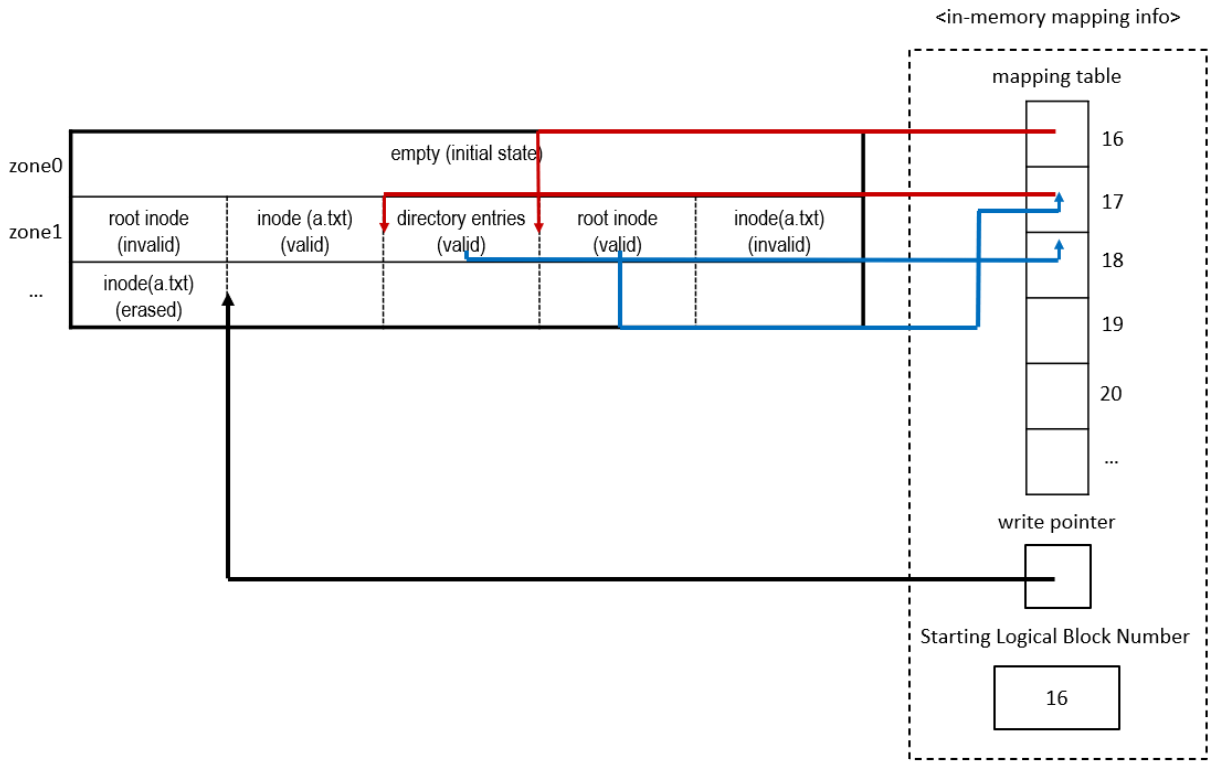
3.2.3. 파일/디렉토리 삭제

파일 삭제는 다음과 같은 세가지 작업이 필요하다.

1. 삭제 대상 파일의 inode를 삭제한다.
2. 삭제 대상 파일의 부모 디렉토리에서, 삭제 대상 파일의 디렉토리 엔트리 데이터를 지운다.
3. 부모 디렉토리의 아이노드에서 자식 파일 개수를 의미하는 변수 값을 1 감소시킨다.

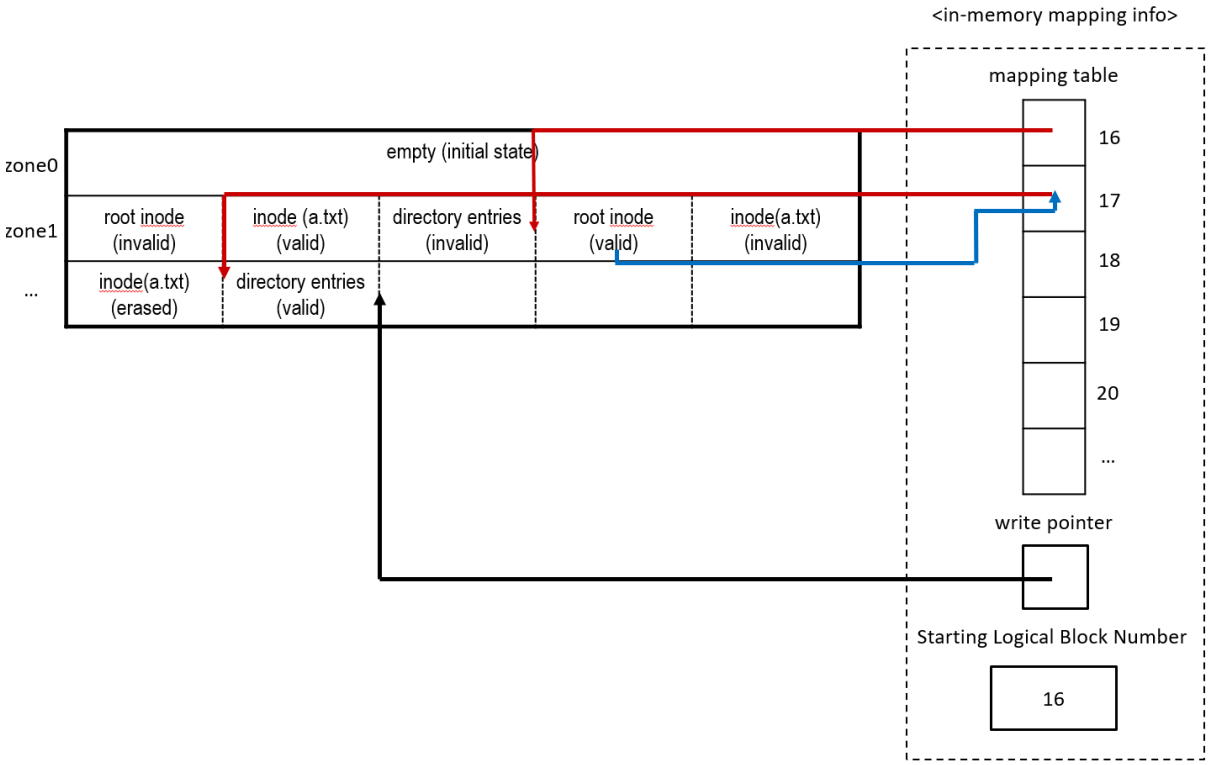
위 작업을 VFS inode와 on-disk inode에 모두 수행해야 하며, 특히 후자의 경우 아이노드나 데이터 블록의 내용에 수정이 발생할 경우, 순차적 쓰기에 의해 기존의 데이터를 invalid 처리하고서, 변경된 대상을 새 위치에 저장해야 한다.

먼저 그림 14와 같이 아이노드를 가리키는 링크 수를 뜻하는 i_nlink를 0으로 설정하여 삭제된 파일의 아이노드임을 표기하도록 한다. 이렇게 수정된 아이노드를 새 위치에 쓰도록 하고, 삭제 대상 파일의 아이노드에게 매핑 테이블에서 할당되어 있던 논리 주소를 해제하고 행을 지우도록 한다.



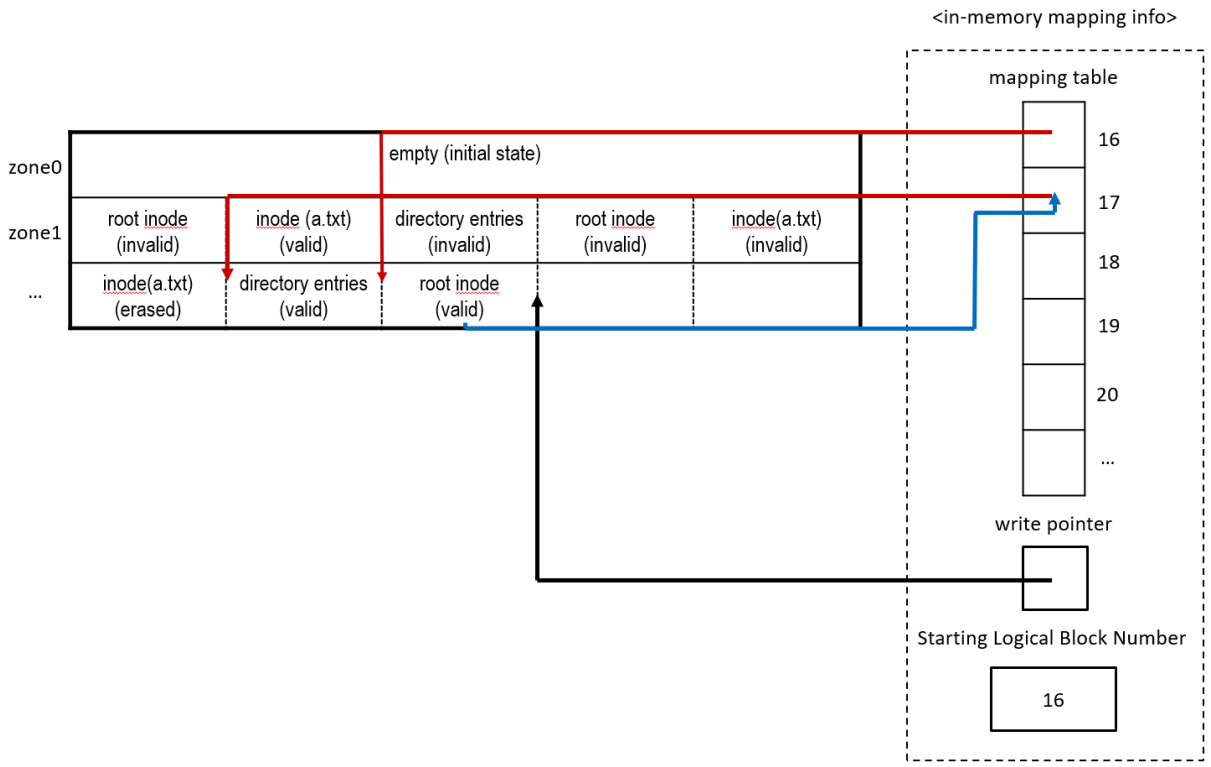
<그림 14: 삭제 대상 파일(/a.txt)의 삭제 표기 후 상태>

다음으로, 부모 디렉토리의 데이터 블록에 저장된 디렉토리 엔트리를 수정하면, 그림 15와 같이 변경된 해당 데이터 블록을 새로운 위치에 쓰고서, 해당 위치 변화를 매핑 테이블에 반영해야 한다.



<그림 15: 루트 디렉토리의 데이터 블록에 쓰여진 디렉토리 엔트리 삭제 후 상태>

부모 디렉토리의 데이터 블록에서 삭제 대상 파일을 가리키는 디렉토리 엔트리를 지웠다면, 그림 16처럼 부모 디렉토리 아이노드의 자식 개수를 의미하는 file_len을 1 감소시켜 반영한다. 그리고서 이 아이노드를 새 위치에 쓰고서, 매핑 테이블에 아이노드의 위치 변화를 반영하도록 한다.



<그림 16: 루트 디렉토리의 아이노드에 자식 파일 개수 감소 후 상태>

디렉토리 삭제는 파일 삭제의 유사한 로직을 가지기에, 해당 함수를 재사용하였다. 다만 디렉토리는 하위 파일이 있을 시 삭제가 일어나면 안 되기에, 디렉토리 아이노드가 가리키는 데이터 블록 속에 디렉토리 엔트리가 쓰여져 있지 않는지를 확인하는 함수를 추가로 두고서, 이를 디렉토리 삭제의 수행 조건으로 사용하였다.

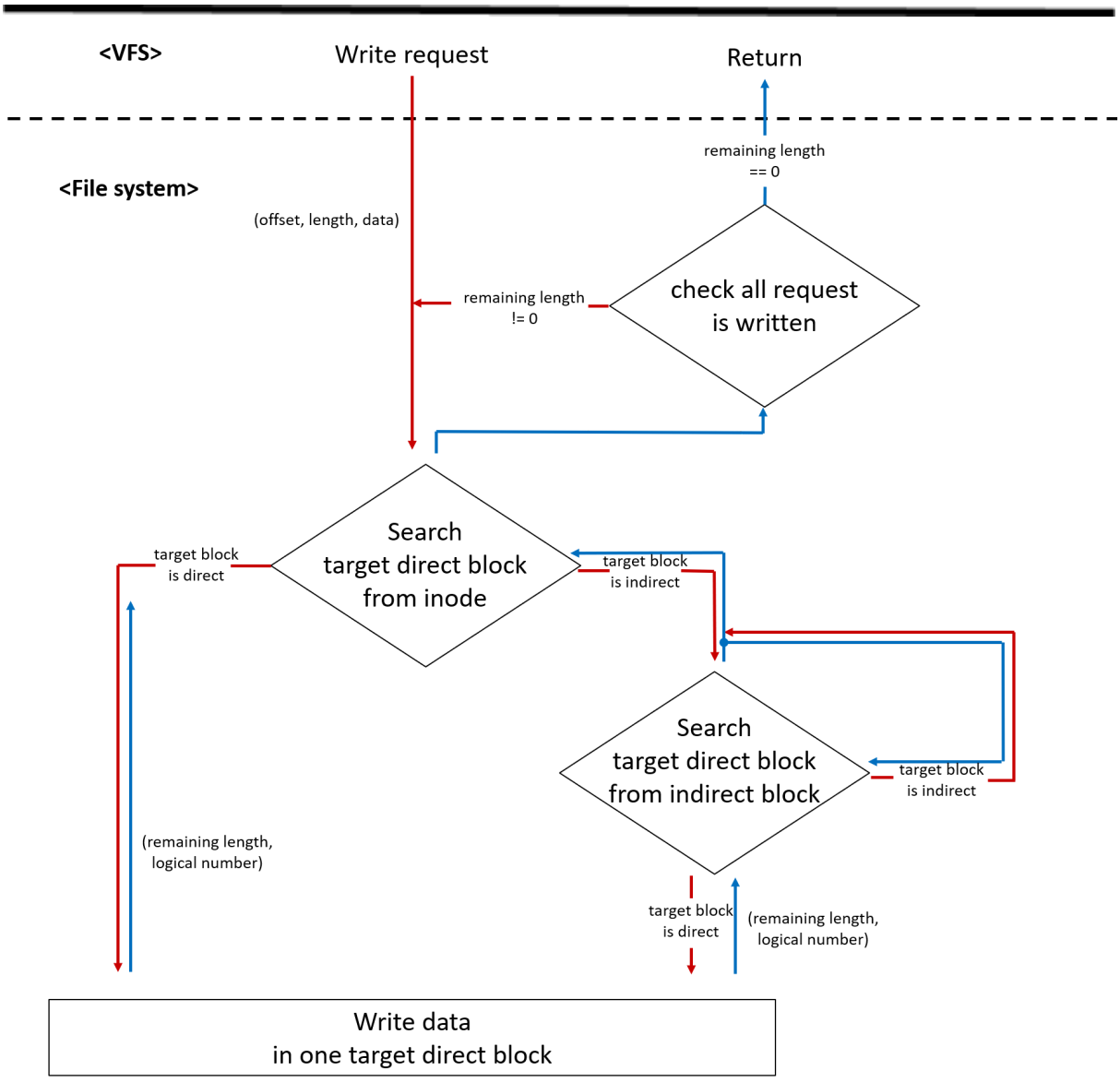
3.2.4. 파일 쓰기

파일 쓰기는 VFS의 쓰기 요청에 담긴 쓰기 오프셋과, 쓰기 길이, 그리고 쓰고자 하는 데이터 세 가지를 바탕으로 이루어진다. 파일시스템은 세 가지 정보를 바탕으로 쓰기를 수행하고서, 실제로 쓰기에 성공한 길이를 VFS로 반환한다.

본 파일시스템에서 아이노드는 direct block 10개와, single indirect block 1개, double indirect block 1개, triple indirect block 1개를 가질 수 있으므로, 이들 중 무엇이 쓰기 요청의 오프셋 지점의 direct block을 포함하고 있는지 계산할 수 있다. 계산 결과의 데이터 블록이 이미 존재한다면 불러오고, 존재하지 않는다면 메모리에 빈 데이터 블록을 생성한다. 실제 데이터의 쓰기가 일어날 direct block에 도달할 때까지 이러한 과정을 재귀적으로 반복한다.

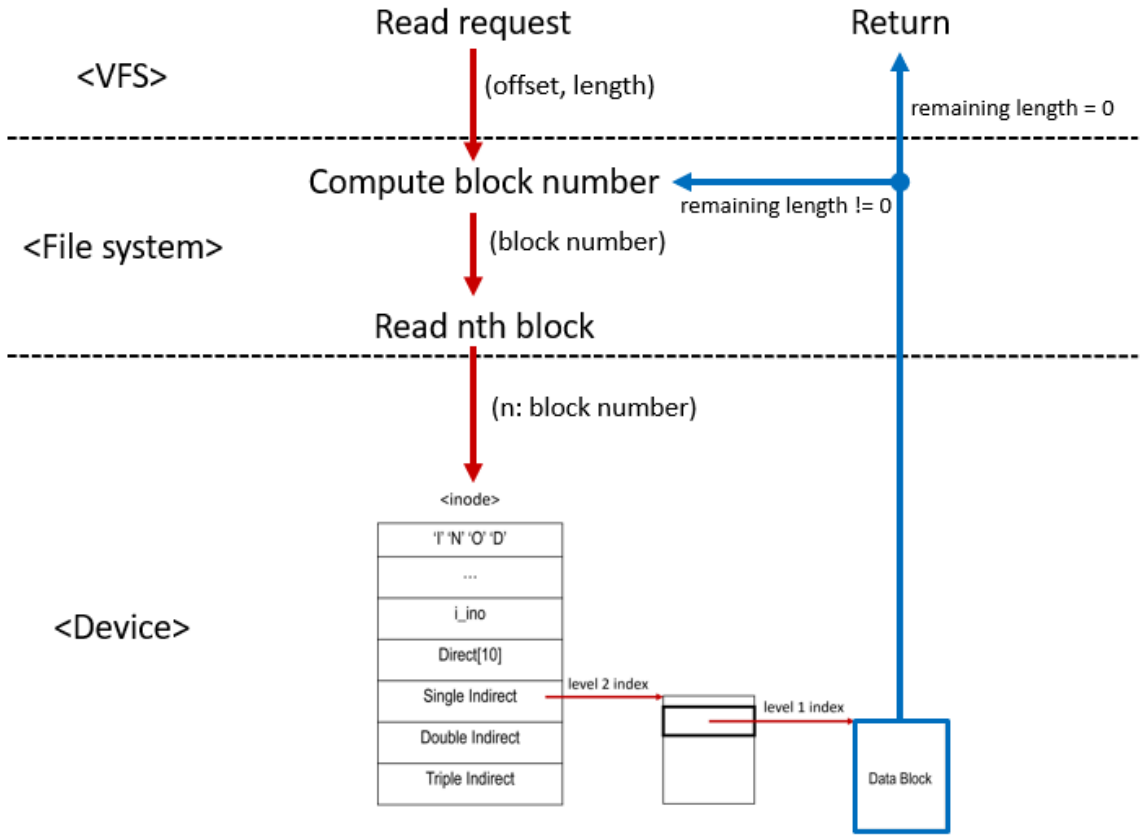
direct block에 도달했다면 기존에 존재하던 데이터들에 쓰기 요청의 데이터를 반영해서 direct block을 업데이트한다. 만일 해당 direct block이 비어져 있었다면 해당 direct block은 위 연산에서 새로 생성된 메모리 상의 데이터 블록이므로, 이후 반환되며 상위 간접 블록이나 아이노드가 이를 가리킬 수 있도록 이번에 새로 할당받은 논리 번호를 출력 매개변수를 통해 반환하도록 한다. 그리고 반환 과정에서 상위 indirect block이나 아이노드는 하위 data block을 가리킬 수 있도록 논리 주소를 자신의 데이터 영역에 기재하도록 한다.

위 쓰기 한 번의 과정으로 하나의 데이터 블록 내에 쓸 수 있는 사용자 데이터 크기 내에서 쓰기가 수행되며, 아직 쓰기 요청 길이가 더 남아있다면 이 과정을 반복 호출하여 다 소진하도록 한다. 이러한 전체 흐름은 그림 17에 나타나 있다.



<그림 17: 파일 쓰기 흐름도>

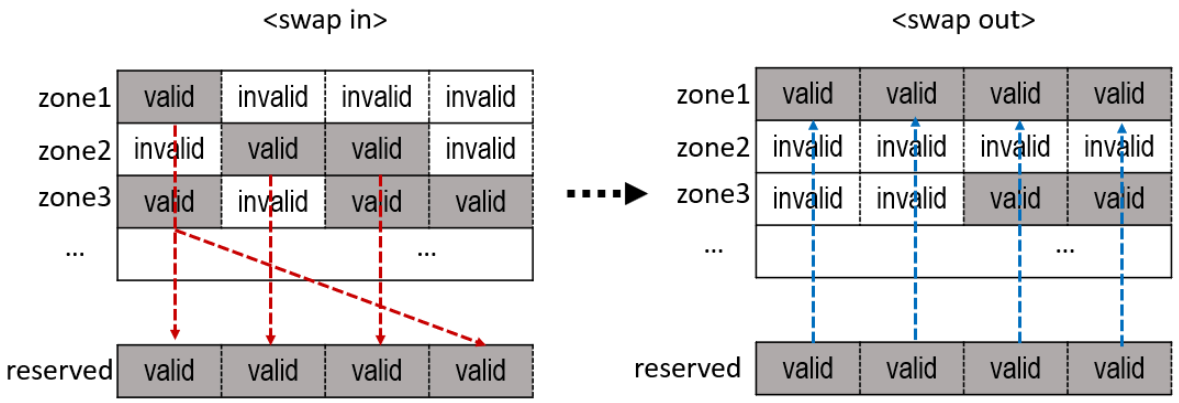
3.2.5. 파일 읽기



<그림 18: 파일 읽기 흐름>

파일 읽기는 그림 18과 같이 VFS로부터 전달된 파일 내 특정 offset과 읽을 길이에 대한 요청을 바탕으로 진행된다. 파일시스템은 먼저 offset이 속한 파일 내 블록 번호를 계산한다. 이어서 해당 블록 번호를 이용해 멀티 레벨 인덱스 블록에 접근한 후, 해당 블록에서부터 요청된 길이를 만족할 때까지 순차적으로 다음 블록에 접근한다. 각 블록의 내용은 iov_iter에 복사되어 상위 계층에 반환된다.

3.2.6. garbage collection



<그림 19: GC swap in, swap out 과정>

garbage collection은 장치의 앞쪽부터 유효한 블록을 읽어 임시 보관소인 reserved zone에 모으는 방식으로 진행된다. 이때 reserved zone에 유효 블록을 모으는 과정을 swap-in, reserved 공간에 모인 블록을 실제 아이노드와 데이터 블록이 위치하는 zone으로 다시 옮기는 과정을 swap-out이라 한다.

garbage collection이 시작되면, 그림 19와 같이 zone 1번부터 garbage collection 비트맵을 확인하여 유효한 데이터 블록을 reserved 블록에 순차적으로 수집(swap-in)한다. reserved 블록이 가득 차면, 앞쪽 zone을 초기화한 뒤 reserved 블록의 데이터를 다시 배치(swap-out)하여 기록한다.

한편, 데이터의 위치가 변하는 swap-in, swap-out 과정에서 매핑 테이블과 garbage collection 비트맵을 상위 계층에서 직접 관리해 줄 필요는 없다. 왜냐하면 장치 입출력 함수가 reserved zone으로 유효한 아이노드나 데이터 블록을 하나씩 읽어오거나, 리셋된 zone에 swap-out 할 때 자동으로 매핑 테이블과 garbage collection 비트맵을 관리해 주기 때문이다.

3.2.7. 기능별 구현 함수

아래 표 1은 사용자 명령어 별로 호출되는 구현 함수들의 목록이고, 표 2는 이 함수들에서 사용하는 도구 함수들의 목록이다.

사용자 명령어	VFS 호출 함수	하위 함수(->)		
		기능 수행 함수	탐색 호출 함수	탐색 수행 함수
mount	hodo_init			
ls	hodo_readdir	hodo_sub_readdir	read_all_dirents	read_all_dirents_from_direct_block read_all_dirents_from_indirect_block
touch	hodo_create			
	hodo_lookup	hodo_sub_lookup	find_inode_number	find_inode_number_from_direct_block find_inode_number_from_indirect_block
mkdir	hodo_mkdir		add_dirent	

사용자 명령어	VFS 호출 함수	하위 함수(->)		
		기능 수행 함수	탐색 호출 함수	탐색 수행 함수
rm	hodo_unlink		remove_dirent	remove_dirent_from_direct_block remove_dirent_from_indirect_block
rmdir	hodo_rmdir	hodo_unlink	check_directory_empty	check_directory_empty_from_direct_block check_directory_emptry_from_indirect_block
echo >	hodo_file_write_iter	hodo_sub_file_write_iter	write_one_block	write_one_block_by_directory_block write_one_block_by_indirect_block
	hodo_setatrr	hodo_sub_setattr		
cat	hodo_file_read_iter		hodo_read_nth_block	
garbage collection	GC			

<표 1: 기능별 구현 함수들>

비트맵용 함수	hodo_get_next_logical_number
	hodo_erase_table_entry
	hodo_get_next_GC_valid
장치 입출력 함수	hodo_read_struct
	hodo_write_struct
	hodo_GC_write_struct
	hodo_GC_read_struct
	compact_datablock
	hodo_read_on_disk_mapping_info
단순 확인 함수	is_dirent_valid
	is_block_logical_number_valid
	is_directblock

<표 2: 도구 함수들>

3.3. 결과 분석 및 평가

3.3.1 테스트 환경 소개

본 과제의 테스트 환경은 표 3에 정리하였다.

호스트 가상환경	WSL
구동 가상환경	FEMU
메모리 크기	4GB
가속화	KVM 가상화 사용
CPU 노출	host CPU 노출
코어 수	가상 CPU 4개 할당
ZNS SSD	존 당 256MB * 16개 존

<표 3: 테스트 환경 소개>

3.3.2 POSIX API 기능 테스트

POSIX API 기능 테스트는 bash 셸 명령어 실행 결과를 통해 검증하였다. 이를 통해 기본적인 POSIX API 지원 여부를 확인할 수 있었으며, 구체적인 명령어와 실행 결과는 표 4에 정리하였다.

파일시스템 마운트	root@ubuntu24:/home/ubuntu# mount | grep /dev/nvme0n1 /dev/nvme0n1 on /mnt type zonefs (rw,relatime,errors=remount-ro)
초기 상태	total 4 16 dr-xr-xr-x 3 root root 3 Sep 13 02:23 . 2 drwxr-xr-x 24 root root 4096 Sep 7 07:29 ..
파일 생성	root@ubuntu24:/mnt# touch a.txt root@ubuntu24:/mnt# ls -ali total 4 16 dr-xr-xr-x 3 root root 4 Sep 13 02:23 . 2 drwxr-xr-x 24 root root 4096 Sep 7 07:29 .. 17 -rw-r--r-- 1 root root 0 Sep 13 02:23 a.txt
파일 쓰기	root@ubuntu24:/mnt# echo aaa > a.txt root@ubuntu24:/mnt# ls -ali total 4 16 dr-xr-xr-x 3 root root 4 Sep 13 02:24 . 2 drwxr-xr-x 24 root root 4096 Sep 7 07:29 .. 17 -rw-r--r-- 1 root root 4 Sep 13 02:24 a.txt
파일 읽기	root@ubuntu24:/mnt# cat a.txt aaa
디렉토리 생성	root@ubuntu24:/mnt# mkdir dir root@ubuntu24:/mnt# ls -ali total 4 16 dr-xr-xr-x 3 root root 4 Sep 13 02:25 . 2 drwxr-xr-x 24 root root 4096 Sep 7 07:29 .. 17 -rw-r--r-- 1 root root 4 Sep 13 02:24 a.txt 20 drwxr-xr-x 1 root root 2 Sep 13 02:25 dir
디렉토리 이동	root@ubuntu24:/mnt# cd dir root@ubuntu24:/mnt/dir# pwd /mnt/dir
디렉토리 내 복수 파일 생성	root@ubuntu24:/mnt/dir# touch a.txt b.txt root@ubuntu24:/mnt/dir# ls -ali total 0 20 drwxr-xr-x 1 root root 4 Sep 13 02:25 . 16 dr-xr-xr-x 3 root root 4 Sep 13 02:25 .. 21 -rw-r--r-- 1 root root 0 Sep 13 02:25 a.txt 23 -rw-r--r-- 1 root root 0 Sep 13 02:25 b.txt
파일 삭제	root@ubuntu24:/mnt/dir# rm a.txt root@ubuntu24:/mnt/dir# ls -ali total 0 20 drwxr-xr-x 1 root root 3 Sep 13 02:26 . 16 dr-xr-xr-x 3 root root 4 Sep 13 02:25 .. 23 -rw-r--r-- 1 root root 0 Sep 13 02:25 b.txt
디렉토리 삭제 (비어있지 않은 경우)	root@ubuntu24:/mnt# pwd /mnt root@ubuntu24:/mnt# ls -ali total 4 16 dr-xr-xr-x 3 root root 4 Sep 13 02:26 . 2 drwxr-xr-x 24 root root 4096 Sep 7 07:29 .. 17 -rw-r--r-- 1 root root 4 Sep 13 02:24 a.txt 20 drwxr-xr-x 1 root root 3 Sep 13 02:26 dir root@ubuntu24:/mnt# rmdir dir rmdir: failed to remove 'dir': Directory not empty
디렉토리 삭제	root@ubuntu24:/mnt# rm -r dir root@ubuntu24:/mnt# ls -ali total 4 16 dr-xr-xr-x 2 root root 3 Sep 13 02:26 . 2 drwxr-xr-x 24 root root 4096 Sep 7 07:29 .. 17 -rw-r--r-- 1 root root 4 Sep 13 02:24 a.txt

<표 4: 기능 시나리오 캡쳐>

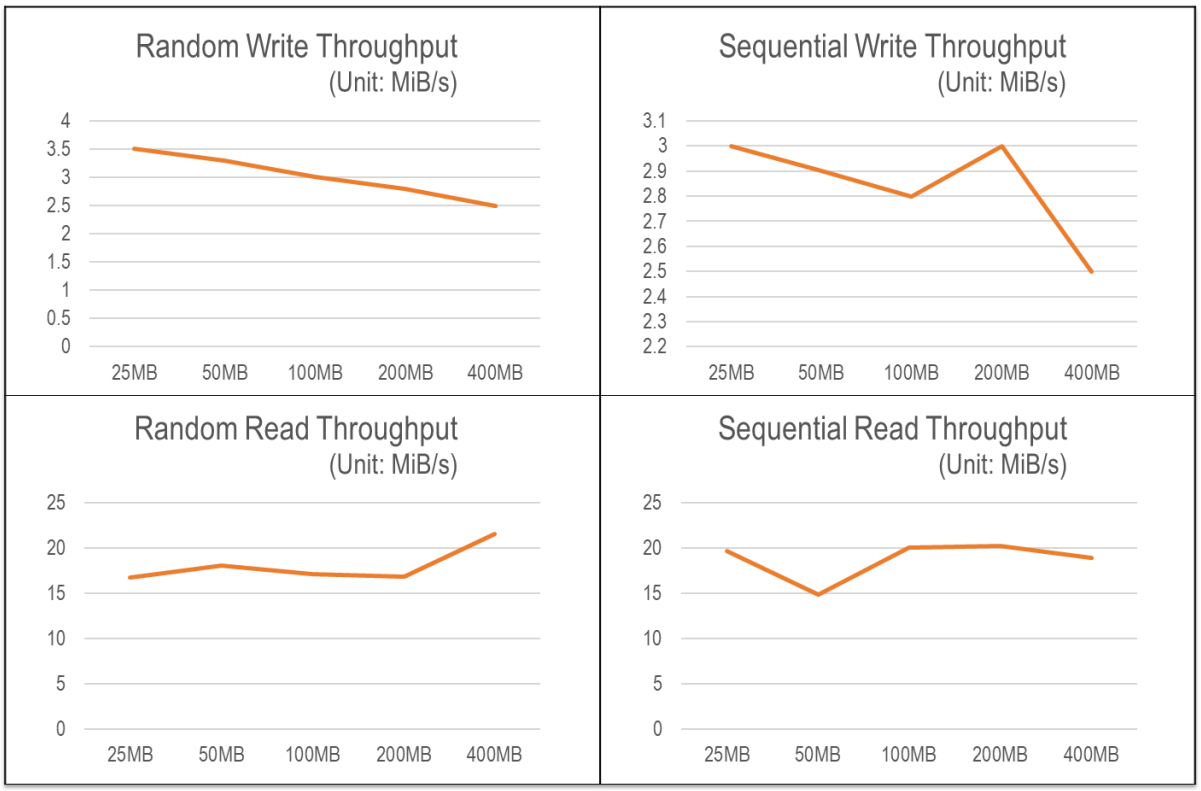
3.3.2. 파일 쓰기-읽기 테스트

파일 쓰기-읽기 테스트는 fio를 이용해서 수행했다. fio는 스토리지나 파일시스템의 성능을 측정하기 위해 다양한 입출력 패턴을 시뮬레이션하는 벤치마크 도구이다.[3]

본 과제에서는 랜덤 쓰기-읽기, 순차 쓰기-읽기 테스트를 진행했다. 표 5에서 테스트에 사용된 fio 명령어를 정리했다. 그림 20에서 각 테스트 수행 결과를 그래프로 정리했다. 파일 쓰기의 경우는 파일의 용량이 증가함에 따라 성능이 감소하는 추세를 보였다. 이는 파일 용량이 커질수록 더 많은 indirect block의 읽기-쓰기 연산이 발생하게 되기 때문으로 해석된다.

Random Write	fio --name=test --filename=/mnt/a.txt --rw=randwrite --bs=4K --size=
Random Read	fio --name=test --filename=/mnt/a.txt --rw=randread --bs=4K --size=
Sequential Write	fio --name=test --filename=/mnt/a.txt --rw=write --bs=4K --size=
Sequential Read	fio --name=test --filename=/mnt/a.txt --rw=read --bs=4K --size=

<표 5: 읽기-쓰기 테스트에 사용된 fio 명령어>



<그림 20: fio 파일 쓰기-읽기 테스트 throughput 측정 그래프>

3.3.3. garbage collection 테스트

본 테스트는 400MB 파일 쓰기를 먼저 수행한 뒤, 이를 garbage collection 하여 수행했다. 표 6과 같이, 400MB 파일 하나를 쓸 경우 실제 장치에는 garbage를 포함해서 3.12GB의 데이터가 쓰였다. 이는 payload 데이터 대비 7.8배의 용량이다. 이 상태에서 garbage collection을 수행하면 표 7에서 나타나듯이, 26초의 시간이 걸려 401.25MB로 장치의 데이터가 수집되었다.

파일 크기	root@ubuntu24:/mnt# ls -ali total 4 16 dr-xr-xr-x 3 root root 4 Sep 13 03:34 . 2 drwxr-xr-x 24 root root 4096 Sep 7 07:29 .. 17 -rw-r--r-- 1 root root 419430400 Sep 13 03:32 a.txt
장치 내 데이터	root@ubuntu24:/mnt/seq# ls -ali total 3932160 18 dr-xr-xr-x 2 root root 15 Sep 13 03:32 . 16 dr-xr-xr-x 3 root root 4 Sep 13 03:34 .. 1 -rw-r----- 1 root root 0 Sep 13 03:32 0 2 -rw-r----- 1 root root 268435456 Sep 13 03:32 1 11 -rw-r----- 1 root root 268435456 Sep 13 03:32 10 12 -rw-r----- 1 root root 268435456 Sep 13 03:32 11 13 -rw-r----- 1 root root 268435456 Sep 13 03:32 12 14 -rw-r----- 1 root root 125722624 Sep 13 03:32 13 15 -rw-r----- 1 root root 0 Sep 13 03:32 14 3 -rw-r----- 1 root root 268435456 Sep 13 03:32 2 4 -rw-r----- 1 root root 268435456 Sep 13 03:32 3 5 -rw-r----- 1 root root 268435456 Sep 13 03:32 4 6 -rw-r----- 1 root root 268435456 Sep 13 03:32 5 7 -rw-r----- 1 root root 268435456 Sep 13 03:32 6 8 -rw-r----- 1 root root 268435456 Sep 13 03:32 7 9 -rw-r----- 1 root root 268435456 Sep 13 03:32 8 10 -rw-r----- 1 root root 268435456 Sep 13 03:32 9

<표 6: garbage collection 수행 전 상태>

GC 실행 시간	root@ubuntu24:/home/ubuntu# ./GCTimeTest GC completed! Elapsed time: 26.341694188 seconds
장치 내 데이터	root@ubuntu24:/mnt/seq# ls -ali total 3932160 18 dr-xr-xr-x 2 root root 15 Sep 13 03:32 . 16 dr-xr-xr-x 3 root root 4 Sep 13 03:36 .. 1 -rw-r----- 1 root root 0 Sep 13 03:32 0 2 -rw-r----- 1 root root 268435456 Sep 13 03:32 1 11 -rw-r----- 1 root root 0 Sep 13 03:32 10 12 -rw-r----- 1 root root 0 Sep 13 03:32 11 13 -rw-r----- 1 root root 0 Sep 13 03:32 12 14 -rw-r----- 1 root root 0 Sep 13 03:32 13 15 -rw-r----- 1 root root 0 Sep 13 03:32 14 3 -rw-r----- 1 root root 152256512 Sep 13 03:32 2 4 -rw-r----- 1 root root 0 Sep 13 03:32 3 5 -rw-r----- 1 root root 0 Sep 13 03:32 4 6 -rw-r----- 1 root root 0 Sep 13 03:32 5 7 -rw-r----- 1 root root 0 Sep 13 03:32 6 8 -rw-r----- 1 root root 0 Sep 13 03:32 7 9 -rw-r----- 1 root root 0 Sep 13 03:32 8 10 -rw-r----- 1 root root 0 Sep 13 03:32 9

<표 7: garbage collection 수행 시간과 수행 후 상태>

3.4. 구성원별 역할 및 개발 일정

구성원별 역할과 개발 일정은 아래의 표 8과 표 9와 같다.

김민준	1. 초기화 기능 구현 2. 파일 생성 기능 구현 3. 디렉토리 생성 기능 구현 4. 파일 속성 변경 기능 구현 5. 파일 읽기 기능 구현
박천휘	1. 파일 존재 확인 기능 구현 2. 디렉토리 내 하위 파일 목록 확인 기능 구현 3. 파일 삭제 기능 구현 4. 디렉토리 삭제 기능 구현 5. 파일 쓰기 기능 구현
공통	1. zonefs 동작 분석 2. 파일시스템 구조 설계 3. GC 관련 설계 개선 및 구현 4. 개발 기록 및 보고서 작성

<표 8: 구성원별 역할>

4월~5월	ZNS SSD 이해
6월	zonefs 분석
7월 초	빌드 환경 준비 및 파일 시스템 탐색 개발
7월 중	파일시스템 구조 자체 설계
7월 말	초기화 및 파일/디렉토리 생성 기능 구현
8월 초	파일/디렉토리 삭제 기능 구현
8월 중	파일 쓰기-읽기 기능 구현
9월 초	GC 관련 설계 개선 및 구현

<표 9: 개발 일정>

4. 결론

4.1. 성과

본 과제의 목표였던 POSIX 파일시스템 API를 제공함으로써 순차적 쓰기만 제공되는 ZNS SSD를 지원하는 파일시스템을 개발했다. 또한, 순차쓰기로 인해 발생하는 garbage를 시간-공간 관점에서 효율적으로 수거하는 garbage collection 기능을 개발하여, 호스트로 이전되었던 garbage collection과 wear leveling에 대한 호스트의 부담을 줄이는데 성공했다.

4.2. 한계 및 향후 개선 방향

본 파일시스템의 한계점은 아래와 같다.

현재 파일 쓰기 구현에서는 direct block에 쓰기 요청 데이터를 반영 후 복귀하는 과정에서 불필요하게 indirect block을 새로 쓰는 문제점이 있다. 추후 구현에서는 indirect block의 업데이트 필요조건을 확인 후 이를 수행하도록 하여 파일 쓰기 연산의 시간 및 공간 효율을 개선하고자 한다.

그리고 현재 설계에서는 zone 1번을 시작으로 데이터 블록이 써진다. 이는 zone 1번에 작업 부하가 과중되는 문제를 야기하고, 결국 zone 1번에 대한 wear level이 높아진다. 따라서 wear leveling을 개선하기 위해서, 쓰기와 garbage collection을 시작하는 zone id를 바꾸어 가며 사용하도록 설계를 개선한다면, 쓰기 부하가 zone에 골고루 분산되어 더 효율적인 wear leveling 관리를 할 수 있을 것이다.

현재 설계에서는 아이노드와 데이터 블록 모두 4KB으로 크기를 통일했다. 그 이유는 외부 단편화를 줄이고자 한 것이다. 하지만, 아이노드의 경우에는 실제로 아이노드의 내용을 담은 크기는 몇백 Bytes 수준이므로 내부 단편화가 발생하는 문제가 있다. 따라서 이를 해결하기 위해서 아이노드를 저장하는 zone과 데이터 블록을 저장하는 zone을 동적으로 각각 구분하여 할당해 관리한다면 내부 단편화 문제를 해결할 수 있을 것으로 기대된다.

5. 참고 문헌

[1] Bjørling, M., Aghayev, A., Holmberg, H., Ramesh, A., Le Moal, D., Ganger, G. R., & Amvrosiadis, G., "Avoiding the block interface tax for flash-based", 2021 USENIX annual technical conference (USENIX ATC 21), pp. 689-703, 2021.

[2] Western Digital, "Zoned Storage File Systems," [Online]. Available: https://zonedstorage.io/docs/filesystems. (Accessed: Sept. 13, 2025)

[3] Jens Axboe. (2017). "fio: Flexible I/O Tester," [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html (Accessed: 2025, Sep. 13)