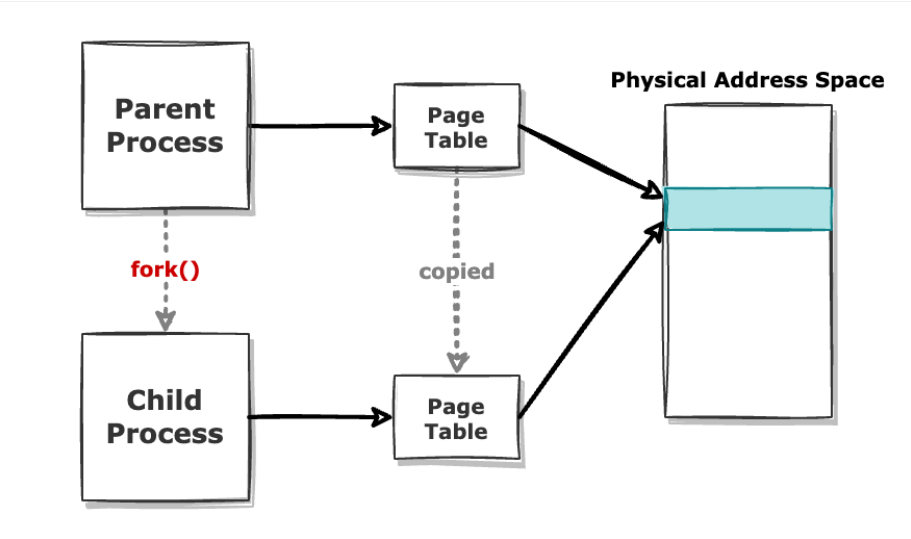
시스템 프로그래밍 과제 1

2020320135 김윤서 (2 free days used)

Copy-on-Write(CoW)의 개념과 원리



Copy-on-Write(CoW)는 컴퓨터 시스템에서 데이터 복사와 쓰기 연산을 최적화하기 위한 기법 중 하나이다. 일반적으로 데이터를 복사할 때는 원본 데이터를 즉시 새 위치에 복제하지만, CoW는 복사 요청이 들어와도 초기에는 실제 복사를 수행하지 않고, 원본 데이터에 대한 참조만을 공유한다. 그러다가 복사본 또는 원본에 수정이 발생하는 시점에, 그 데이터 블록을 새로운 위치에 복사하여 수정하는 방식이다. 즉, 쓰기(write) 시점에만 복사가 발생하는 구조로, 이름 그대로 쓰기 시 복사(Copy on Write) 방식이다.

CoW는 메모리 영역뿐만 아니라 파일 시스템에도 적용되며, 성능 향상과 디스크 공간 절약이라는 큰 장점을 제공한다. 예를 들어, 대용량 파일을 복사할 때 일반 파일 시스템에서는 데이터 전체를 새로운 블록에 복제하는 데 반해, CoW 기반 파일 시스템은 초기 복사 시점에는 메타데이터만 처리하고, 실제 데이터 블록은 공유한다. 이후 복사본에 쓰기가 발생하면 그 시점에서야 별도의 블록을 할당한다. 이 방식은 특히 가상화, 컨테이너 기술, 백업/스냅샷 시스템 등에서 큰 이점을 제공하며, 효율적인 시스템 자원 관리가 가능해진다.

파일 시스템 관점에서 CoW는 크게 두 가지 영역에 적용된다. 첫째는 파일 데이터 블록이며, 둘째는 메타데이터(inode, 디렉토리 엔트리 등)이다. CoW가 메타데이터에도 적용될 경우, 디렉토리 구조나 inode 수정 시에도 원본 메타데이터를 그대로 두고 새로운 구조만 생성하므로, 스냅샷과 롤백 기능을 매우 간단히 구현할 수 있다고 한다. 대표적인 CoW 파일 시스템으로는 Btrfs, ZFS, APFS(Apple), F2FS 등이 있다.

EXT4 파일 시스템

EXT4(Extended Filesystem 4)는 Linux에서 가장 널리 사용되는 고전적인 파일 시스템이며, ext3의 후속 버전으로 안정성과 성능 모두를 향상시킨 구조를 갖는다. ext4는 저널링(journaling) 기능을 통해 데이터 일관성을 보장하며, 주요 특징으로는 extents(연속 블록), delayed allocation, multiblock allocation, 빠른 fsck, persistent preallocation 등을 들 수 있다.

EXT4는 기본적으로 CoW를 지원하지 않는다. 즉, 파일을 복사하거나 수정하면 항상 새로운 데이터 블록을 할당하고, 기존 블록은 덮어쓰거나 유지된다. 이로 인해 대용량 파일 복사 시 실제 데이터가 디스크 상에서 복제되어, I/O와 저장 공간이 모두 추가로 소모된다. 이러한 구조는 Btrfs와 같은 CoW 기반 파일 시스템에 비해 복사 및 백업 시 비효율적이지만, 일관성과 예측 가능한 동작 방식으로 인해 많은 서버와 임베디드 시스템에서 여전히 사용되고 있다고 한다.

EXT4의 쓰기 연산은 보통 페이지 캐시를 활용한 Buffered I/O 방식으로 처리된다. 사용자가 write() 시스템 콜을 호출하면, 데이터는 커널의 페이지 캐시(page cache)에 기록되고, 해당 페이지는 dirty page로 마킹된다. 이 dirty page는 특정 주기마다, 또는 fsync() 호출 시 ext4_writepages() 함수를 통해 디스크에 flush된다.

BTRFS 파일 시스템

Btrfs(B-tree File System)는 차세대 리눅스 파일 시스템으로, CoW 를 기본 설계로 채택한 파일 시스템이다. Btrfs 는 ext4 와 달리 파일 데이터뿐만 아니라 모든 메타데이터도 CoW 방식으로 처리하여 스냅샷, 롤백, 중복 제거, 압축, RAID 기능 등을 기본으로 제공한다.

Btrfs 에서 파일 복사는 cp --reflink=auto 또는 cp --reflink=always 명령으로 수행되며, 이 경우 실제 데이터 블록은 복사되지 않고 원본 파일의 블록을 공유하게 된다. 복사된 파일이 수정되지 않는 한 디스크 상에는 실제로 하나의 데이터 블록만 존재하며, 공간 효율이 매우 높다. 만약 복사본에 write()가 수행되면, 해당 블록에 대한 CoW 가 즉시 발동되어 새로운 블록이 할당되고 그곳에 데이터가 기록된다. 이 구조로 인해 복사 성능이 매우 우수하며, I/O 최소화가 가능하다.

Btrfs 의 쓰기 과정은 페이지 캐시에 데이터가 저장된 후, 기존 블록을 변경하는 대신 새로운 블록을 할당하고 그 블록에 데이터를 쓴 뒤, 메타데이터 포인터를 새로운 위치로 갱신하는 구조이다. 이렇게 하면 기존 데이터는 유지되므로, 시스템 장애나 비정상 종료 상황에서도 데이터를 보호할 수 있다. 이를 통해 ext4 의 journaling 보다 안전하게 데이터를 보호할 수 있다.

VFS란

VFS(Virtual File System)는 리눅스 커널 내에서 여러 종류의 파일 시스템(ext4, btrfs 등) 을 공통된 인터페이스로 추상화해주는 계층이다. 즉, 사용자는 파일 시스템 종류에 관계없이 open(), read(), write() 같은 시스템 콜을 동일하게 사용할 수 있고, VFS는 이를 각 파일 시스템에 맞게 연결해주는 중간 계층 역할을 한다.

파일 시스템 공통 구조체

구조체	정의 파일	역할
struct file_operations	include/linux/fs.h	파일 단위 연산 (read, write, open 등)
struct inode_operations	include/linux/fs.h	inode 기반 연산 (create, lookup, link 등)
struct super_operations	include/linux/fs.h	파일 시스템 전반 제어 (mount/unmount 등)
struct address_space_operations	include/linux/fs.h	페이지 캐시와 관련된 연산 (writepage 등)
struct dentry_operations	include/linux/dcache.h	디렉토리 엔트리 연산

1. struct file_operations

파일 객체(struct file)에 연결되며, 열린 파일에 대해 가능한 작업들을 정의

사용 예시) write() 호출 시 → file->f_op->write_iter() 호출됨

2. struct inode_operations

파일의 메타데이터를 조작하는 함수 집합

사용 예시) 디렉토리 내부에 새로운 파일을 생성할 때 → inode->i_op->create()

3. struct super_operations

하나의 파일 시스템 전체를 관리하는 연산을 처리

사용 예시) 파일 시스템 마운트 시 → super_block->s_op->alloc_inode() 등 호출

4. struct address_space_operations

페이지 캐시와 디스크 간의 데이터 전송 담당

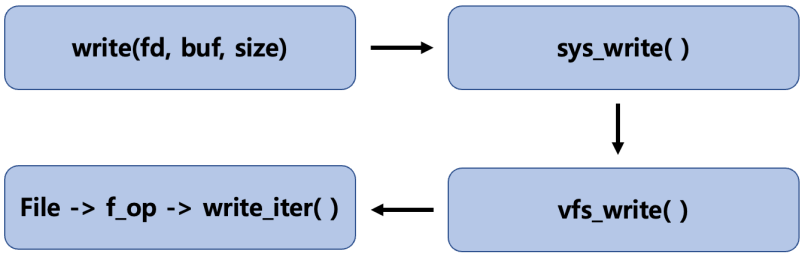
사용 예시) dirty page를 flush할 때 → mapping->a_ops->writepage() 호출

5. struct dentry_operations

dentry (디렉토리 엔트리)와 관련된 동작 정의

사용 예시) 디렉토리 엔트리를 다시 유효성 검사할 때 → dentry->d_op->d_revalidate()

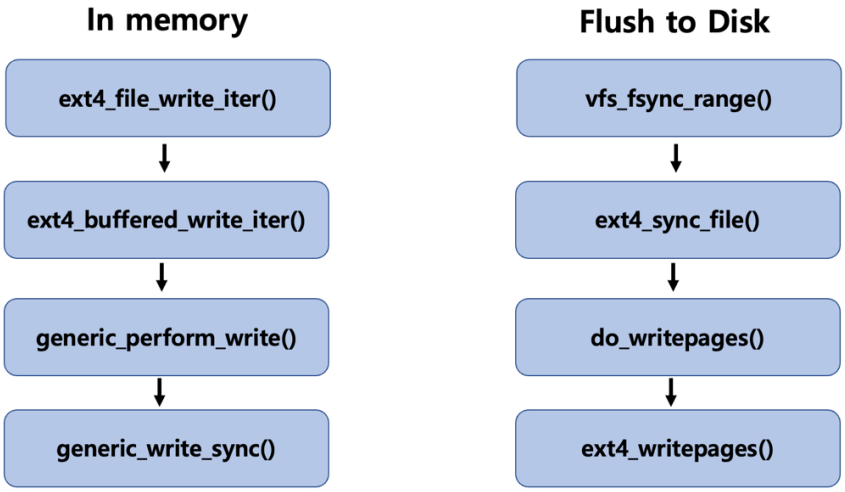
파일 복사(copy)시 파일시스템 간 공통 흐름



파일을 복사할 때, 리눅스 커널 내부에서는 VFS(Virtual File System)를 중심으로 여러 파일 시스템에서 공통적으로 동작하는 흐름이 존재한다. 사용자가 cp 명령어를 실행하거나 write() 시스템 콜을 호출하면, 해당 요청은 먼저 sys_write()를 통해 커널로 진입하고, 이어서 vfs_write() 함수로 전달된다. 이 시점까지는 ext4, btrfs, xfs 등 어떤 파일 시스템을 사용하더라도 흐름은 동일하다.

VFS 는 열린 파일 객체(struct file)의 file_operations 구조체에 정의된 write_iter() 함수를 호출하여, 각 파일 시스템에 실제 쓰기 동작을 맡긴다. 이때부터 파일 시스템별 차이가 발생한다. 예를 들어 ext4 는 ext4_file_write_iter() 함수를, btrfs 는 btrfs_file_write_iter() 함수를 통해 각각의 방식으로 데이터를 처리한다.

EXT4 에서의 과정

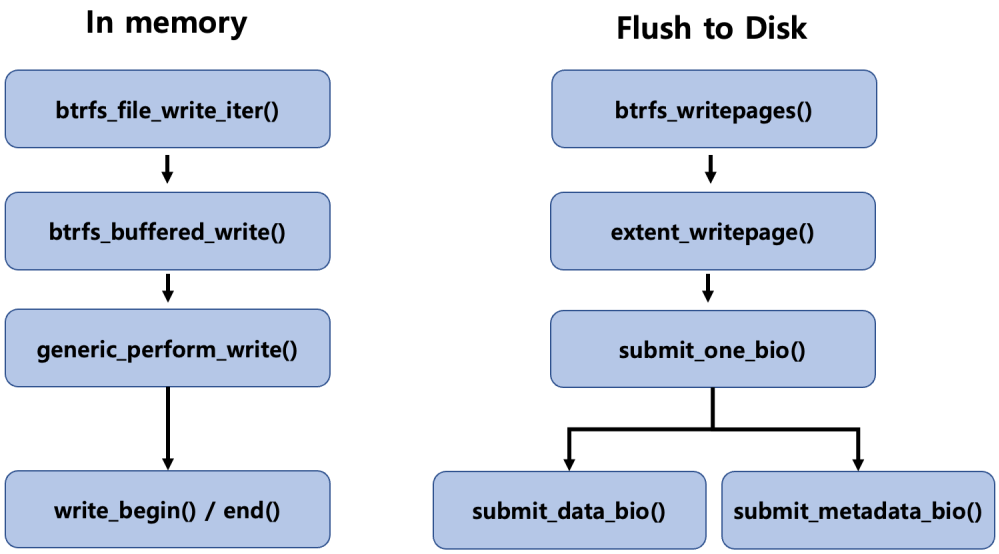


사용자가 새로운 파일을 생성하고 내용을 쓰면, 먼저 해당 데이터는 디스크에 직접 기록되지 않고 메모리의 페이지 캐시(Page Cache)에 저장된다. 이때 커널은 ext4_file_write_iter()를 통해 파일 시스템 write 경로로 진입하며, generic_perform_write()를 통해 해당 데이터를 메모리에 기록하고 해당 페이지를 Dirty Page로 표시한다. 즉, 이 시점까지는 모든 데이터가 메모리에서만 처리되며, 실제 디스크

에는 아무런 변화가 없다. 파일을 닫거나 fsync() 요청이 발생하면, ext4_writepages()를 통해 Dirty Page가 디스크로 flush되어 비로소 파일 내용이 디스크에 저장된다.

사용자가 기존 파일을 복사하면, 커널은 먼저 원본 파일의 내용을 디스크에서 읽거나 이미 캐시에 있는 경우 이를 활용한다. 복사된 새 파일 역시 메모리의 페이지 캐시에 먼저 작성되며 Dirty Page로 등록된다. 이때 생성되는 것은 새로운 메타데이터(inode, dentry 등), 복사된 파일의 데이터이다. 따라서 파일 복사 과정 역시 처음에는 모두 메모리에서 수행되며, 이후 fsync() 호출이나 파일이 닫힐 때 Dirty Page가 ext4_writepages()를 통해 디스크로 flush되어 복사본이 디스크에 반영된다.

BTRFS 에서의 과정



사용자가 새로운 파일을 생성하면, 먼저 해당 파일의 데이터는 메모리의 페이지 캐시에 저장된다. 이때 btrfs_file_write_iter()를 통해 데이터가 기록되며, 내부적으로 generic_perform_write()를 통해 페이지 캐시에 적재된 데이터는 Dirty Page 로 마킹된다. 이 시점까지는 모든 처리가 메모리에서 이루어지며, 실제 디스크에는 어떤 데이터도 기록되지 않는다. 이후 fsync() 호출이나 파일이 닫히는 시점에서 btrfs_writepages()가 호출되어 Dirty Page 가 디스크로 flush 된다. 이 과정에서 데이터 블록은 CoW 방식으로 새롭게 할당된 영역에 기록되며, 동시에 해당 블록을 참조하는 inode 나 extent 정보 등 메타데이터도 함께 디스크에 반영된다.

이후 사용자가 해당 파일을 복사할 경우(cp --reflink=auto), Btrfs 는 Copy-on-Write 방식을 활용하여 원본 파일의 데이터 블록을 그대로 공유하고, 새로운 파일을 위한 inode, 디렉토리 엔트리, extent 항목 등의 메타데이터만 생성한다.

소스코드

usr/src/linux-5.15/fs/ext4/page-io.c

```
// 로그
if (dentry) {
    char *buf = (char *)__get_free_page(GFP_KERNEL);
    if (buf) {
        char *path = dentry_path_raw(dentry, buf, PAGE_SIZE);
        if (!IS_ERR(path)) {
            // "test"가 경로에 포함되어 있을 때 && offset이 512KB 단위일 때만 출력
            if ((strstr(path, "test") != NULL) && (offset % (1024 * 512) == 0)) {
                printk(KERN_INFO "[EXT4][BIO_WRITE] file=%s inode=%lu offset=%llu\n",
                       path, inode->i_ino, (unsigned long long)offset);
            }
            // if (strstr(path, "test") != NULL) {
            //  printk(KERN_INFO "[EXT4][BIO_WRITE] file=%s inode=%lu offset=%llu\n",
            //         path, inode->i_ino, (unsigned long long)offset);
            // }
        }
        free_page((unsigned long)buf);
    }
    dput(dentry);
}
```

설명

EXT4 파일 시스템의 실제 디스크 쓰기(bio write)가 발생하는 시점인 ext4_bio_write_page() 함수에 로그 출력을 추가하였다. d_find_alias()를 통해 inode에 대응되는 dentry를 찾고, dentry_path_raw()를 사용해 파일 경로를 얻은 뒤, 경로에 "test"라는 문자열이 포함되어 있고 offset이 512KB 단위일 경우에만 로그를 출력하도록 조건을 걸었다. offset은 현재 페이지의 index를 기준으로 계산하였으며, 로그는 [EXT4][BIO_WRITE] file=경로 inode=번호 offset=값 형태로 출력되어 디스크에 flush되는 시점의 파일 경로, inode 번호, offset 정보를 확인할 수 있다.

usr/src/linux-5.15/fs/btrfs/extent_io.c

```
// 로그
offset = page_offset(page);
dentry = d_find_alias(inode);
if (dentry) {
    buf = (char *)__get_free_page(GFP_KERNEL);
    if (buf) {
        path = dentry_path_raw(dentry, buf, PAGE_SIZE);
        if (!IS_ERR(path)){
            // "test"가 경로에 포함되어 있을 때 &&offset이 512KB 단위일 때만 출력
            if ((strstr(path, "test") != NULL) && (offset % (1024 * 512) == 0)){
                printk(KERN_INFO "[BTRFS][WRITEPAGE] file=%s inode=%lu offset=%llu\n",
                    path, inode->i_ino, (unsigned long long)offset);
            }
            // if (strstr(path, "test") != NULL){
            //  printk(KERN_INFO "[BTRFS][WRITEPAGE] file=%s inode=%lu offset=%llu\n",
            //      path, inode->i_ino, (unsigned long long)offset);
            // }
        }
        free_page((unsigned long)buf);
    }
    dput(dentry);
}
```

설명

Btrfs 파일 시스템에서 CoW 방식으로 디스크에 flush가 발생하는 시점인 __extent_writepage() 함수에 로그 출력을 추가하였다. 페이지의 파일 내 위치(offset)는 page_offset()을 통해 계산하였고, 해당 페이지가 어떤 파일에 속하는지를 확인하기 위해 d_find_alias()로 dentry를 구한 후, dentry_path_raw()를 이용해 경로 문자열을 얻었다. 이후 해당 경로에 "test"가 포함되어 있고 offset이 정확히 512KB 단위일 때만 로그를 출력하도록 조건을 설정하였다. 출력되는 로그는 [BTRFS][WRITEPAGE] file=경로 inode=번호 offset=값 형식이며, 이를 통해 특정 실험 대상 파일의 디스크 flush 시점, inode 번호, 쓰기 offset을 명확하게 추적할 수 있다.

테스트 환경

가상 디스크 생성 및 마운트

```
sudo umount /home/kys/ext4_dir
sudo umount /home/kys/btrfs_dir
sudo losetup -D

cd ~
dd if=/dev/zero of=ext4file count=1 bs=2G
dd if=/dev/zero of=btrfsfile count=1 bs=2G

mkfs.ext4 ./ext4file
mkfs.btrfs ./btrfsfile
sudo losetup /dev/loop12 ext4file
sudo losetup /dev/loop13 btrfsfile
sudo mount -t ext4 /dev/loop12 ~/ext4_dir
sudo mount -t btrfs /dev/loop13 ~/btrfs_dir
```

기존에 마운트되어 있던 EXT4와 Btrfs 디렉토리(/home/kys/ext4_dir, /home/kys/btrfs_dir)를 umount 명령어로 마운트 해제하고, losetup -D를 통해 기존에 설정된 루프 디바이스(loop device)를 모두 해제한다.

이후 홈 디렉토리에서 dd 명령어를 사용하여 2GB 크기의 빈 파일(ext4file, btrfsfile)을 각각 생성한다. 이 파일들은 가상 디스크의 역할을 하며, 이후 각각 EXT4와 Btrfs 파일 시스템으로 포맷된다. mkfs.ext4와 mkfs.btrfs 명령어로 각각의 파일을 EXT4와 Btrfs 형식으로 포맷한 후, losetup 명령어를 통해 이 파일들을 /dev/loop12와 /dev/loop13에 연결하여 루프 디바이스로 설정한다.

마지막으로 mount 명령어를 사용해 /dev/loop12를 EXT4 파일 시스템으로, /dev/loop13을 Btrfs 파일 시스템으로 각각 마운트하고, 해당 디렉토리(~/ext4_dir, ~/btrfs_dir)에서 사용할 수 있도록 준비한다.

테스트 스크립트

/home/kys/testscriopt_ext4.sh

```bash
#!/bin/bash

home_dir="/home/kys/ext4_dir"

echo "[INFO] Cleaning up old files in ${home_dir}..."
sudo rm -rf ${home_dir}/*

echo "[INFO] Creating initial 2MB test file..."
sudo dd if=/dev/zero of=${home_dir}/test bs=1024 count=2000 status=progress

echo "[INFO] Starting copy operations..."

# 파일 100번 복사
for i in $(seq 1 100); do
    sudo cp ${home_dir}/test ${home_dir}/test_$i
    # 복사 진행 로그 출력
    if (( $i % 10 == 0 )); then
        echo "[INFO] Copied $i files to ${home_dir}"
    fi
done

echo "[INFO] Copy complete!"
```

설명

먼저 해당 디렉토리의 기존 파일을 모두 삭제한 뒤, dd 명령어를 사용해 2MB 크기의 초기 파일 test를 생성한다. 이후 이 파일을 test_1부터 test_100까지 100회 복사하며, 10개 단위로 복사 진행 상황을 출력한다. 복사에는 --reflink 옵션이 사용되지 않아, 매 복사마다 실제 데이터 블록이 디스크에 기록되며 inode도 새로 할당된다.

/home/kys/testscript_btrfs.sh

```bash
#!/bin/bash

home_dir="/home/kys/btrfs_dir"

echo "[INFO] Cleaning up old files in ${home_dir}..."
sudo rm -rf ${home_dir}/*

echo "[INFO] Creating initial 2MB test file..."
sudo dd if=/dev/zero of=${home_dir}/test bs=1024 count=2000 status=progress

echo "[INFO] Starting copy operations..."

# 파일 100번 복사
for i in $(seq 1 100); do
    sudo cp --reflink=auto ${home_dir}/test ${home_dir}/test_$i
    # 복사 진행 상황 출력
    if (( $i % 10 == 0 )); then
        echo "[INFO] Copied $i files to ${home_dir}"
    fi
done

echo "[INFO] Copy complete!"
```

설명

스크립트 실행 시 먼저 해당 디렉토리 내 기존 파일들을 모두 삭제하고, dd 명령어를 사용해 2MB 크기의 테스트 파일 test를 생성한다. 이후 이 파일을 test_1부터 test_100까지 100회 복사하는데, cp 명령에 --reflink=auto 옵션을 사용함으로써 Btrfs의 CoW 특성이 활성화된다. 이 방식은 파일 내용이 변경되지 않는 한 동일한 데이터 블록을 참조하며, 복사 과정 중에는 10개 단위로 복사 진행 상황을 출력하여 진행 상태를 확인할 수 있다.

테스트 스크립트 실행

```
# 실행 권한 설정
chmod +x testscript_ext4.sh
chmod +x testscript_btrfs.sh

# 테스트 스크립트 실행
sudo ./testscript_ext4.sh
sudo ./testscript_btrfs.sh
```

로그 확인

```
# ext4 로그
sudo dmesg -C
sudo dmesg -wH | grep -i "ext4"

# btrfs 로그
sudo dmesg -C
sudo dmesg -wH | grep -i "btrfs"
```

실험 결과 - ext4

```
kys@2020320135:~$ sudo dmesg -C
sudo dmesg -wH | grep -i "ext4"
[   +8.194293] [EXT4][MPAGE_PREPARE_EXTENT_TO_MAP] file=/test inode=11 offset=0
[   +0.000018] [EXT4][WRITE_PAGES] file=/test inode=11 offset=17617849730505637888
[   +0.000008] [EXT4][MPAGE_PREPARE_EXTENT_TO_MAP] file=/test inode=11 offset=0
[   +0.000136] [EXT4][MPAGE_MAP_AND_SUBMIT_EXTENT] file=/test inode=11 logical_offset=0 blocks=500
[   +0.000003] [EXT4][MPAGE_MAP_AND_SUBMIT_BUFFERS] file=/test inode=11 offset=0
[   +0.000003] [EXT4][MPAGE_SUBMIT_PAGE] file=/test inode=11 offset=0 len=4096
[   +0.000004] [EXT4][BIO_WRITE] file=/test inode=11 offset=0
[   +0.000069] [EXT4][MPAGE_MAP_AND_SUBMIT_BUFFERS] file=/test inode=11 offset=524288
[   +0.000001] [EXT4][MPAGE_SUBMIT_PAGE] file=/test inode=11 offset=524288 len=4096
[   +0.000001] [EXT4][BIO_WRITE] file=/test inode=11 offset=524288
[   +0.000062] [EXT4][MPAGE_MAP_AND_SUBMIT_BUFFERS] file=/test inode=11 offset=1048576
[   +0.000001] [EXT4][MPAGE_SUBMIT_PAGE] file=/test inode=11 offset=1048576 len=4096
[   +0.000001] [EXT4][BIO_WRITE] file=/test inode=11 offset=1048576
[   +0.000060] [EXT4][MPAGE_MAP_AND_SUBMIT_BUFFERS] file=/test inode=11 offset=1572864
[   +0.000001] [EXT4][MPAGE_SUBMIT_PAGE] file=/test inode=11 offset=1572864 len=4096
[   +0.000002] [EXT4][BIO_WRITE] file=/test inode=11 offset=1572864
[   +0.000106] [EXT4][MPAGE_PREPARE_EXTENT_TO_MAP] file=/test_1 inode=12 offset=0
[   +0.000003] [EXT4][MPAGE_PREPARE_EXTENT_TO_MAP] file=/test_1 inode=12 offset=0
[   +0.000053] [EXT4][MPAGE_MAP_AND_SUBMIT_EXTENT] file=/test_1 inode=12 logical_offset=0 blocks=500
[   +0.000002] [EXT4][MPAGE_MAP_AND_SUBMIT_BUFFERS] file=/test_1 inode=12 offset=0
[   +0.000001] [EXT4][MPAGE_SUBMIT_PAGE] file=/test_1 inode=12 offset=0 len=4096
[   +0.000002] [EXT4][BIO_WRITE] file=/test_1 inode=12 offset=0
[   +0.000063] [EXT4][MPAGE_MAP_AND_SUBMIT_BUFFERS] file=/test_1 inode=12 offset=524288
[   +0.000001] [EXT4][MPAGE_SUBMIT_PAGE] file=/test_1 inode=12 offset=524288 len=4096
[   +0.000001] [EXT4][BIO_WRITE] file=/test_1 inode=12 offset=524288
[   +0.000065] [EXT4][MPAGE_MAP_AND_SUBMIT_BUFFERS] file=/test_1 inode=12 offset=1048576
[   +0.000001] [EXT4][MPAGE_SUBMIT_PAGE] file=/test_1 inode=12 offset=1048576 len=4096
[   +0.000002] [EXT4][BIO_WRITE] file=/test_1 inode=12 offset=1048576
[   +0.000061] [EXT4][MPAGE_MAP_AND_SUBMIT_BUFFERS] file=/test_1 inode=12 offset=1572864
[   +0.000001] [EXT4][MPAGE_SUBMIT_PAGE] file=/test_1 inode=12 offset=1572864 len=4096
[   +0.000001] [EXT4][BIO_WRITE] file=/test_1 inode=12 offset=1572864
```

해당 로그는 EXT4 파일 시스템에서 /test를 생성 후 /test_1 파일을 복사하고 디스크에 flush되는 과정을 보여준다. 먼저, 파일 /test에 대해 페이지 매핑을 준비하고(extent 할당), 페이지를 더티 처리한 후 mpage_map_and_submit_extent, mpage_submit_page 등을 통해 디스크 쓰기를 준비한다.

파일 /test_1 역시 같은 방식으로 처리되며, inode만 12번으로 다를 뿐 전체 흐름은 /test와 동일하게 반복된다. 즉, EXT4는 복사 시 파일 내용을 메모리에 적재하고, dirty 페이지로 표시한 뒤, 디스크에 페이지 단위로 flush하는 구조다. 이 과정에서 파일 크기가 클수록 flush가 반복되며 디스크 쓰기 횟수도 많아진다. 이후 /test_100까지 해당 과정을 반복한다.

실험 결과 – btrfs

```
kys@2020320135:~$ sudo dmesg -C
sudo dmesg -wH | grep -i "btrfs"
[Apr19 02:44] [BTRFS][WRITEPAGES] file=/test inode=257 writeback start
[  +0.000005] [BTRFS][EXTENT_WRITEPAGES] file=/test inode=257 writeback triggered
[  +0.000003] [BTRFS][EXTENT_WRITEPAGE] file=/test inode=257 offset=0
[  +0.000130] [BTRFS][EXTENT_WRITEPAGE] file=/test inode=257 offset=524288
[  +0.000050] [BTRFS][EXTENT_WRITEPAGE] file=/test inode=257 offset=1048576
[  +0.000001] [BTRFS][SUBMIT_EXTENT_PAGE] file=/test inode=257 offset=1048576 size=4096 [type=data]
[  +0.000001] [BTRFS][SUBMIT_ONE_BIO][SYNC] ino=18173366821440341633, sector=26624, size=1048576
[  +0.000001] [BTRFS][SUBMIT_DATA_BIO] inode=257, offset=13631488, size=1048576
[  +0.000061] [BTRFS][EXTENT_WRITEPAGE] file=/test inode=257 offset=1572864
[  +0.000057] [BTRFS][SUBMIT_ONE_BIO][SYNC] ino=18173366821440341633, sector=28672, size=999424
[  +0.000001] [BTRFS][SUBMIT_DATA_BIO] inode=257, offset=14680064, size=999424
[  +0.000133] [BTRFS][MAP_BIO] logical=13631488, len=1048576, op=1, devs=1
[  +0.000003] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=13631488, bio_sectors=26624, op=1
[  +0.000103] [BTRFS][MAP_BIO] logical=14680064, len=999424, op=1, devs=1
[  +0.000002] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=14680064, bio_sectors=28672, op=1
[  +2.126862] [BTRFS][SUBMIT_EXTENT_PAGE] inode=1 offset=30474240 size=4096 [type=meta] (no dentry)
[  +0.000033] [BTRFS][SUBMIT_ONE_BIO][SYNC] ino=13009195992709632625, sector=59456, size=32768
[  +0.000129] [BTRFS][MAP_BIO] logical=30441472, len=32768, op=1, devs=2
[  +0.000007] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=38830080, bio_sectors=59456, op=1
[  +0.000039] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=146178048, bio_sectors=59456, op=1
[  +0.000012] [BTRFS][SUBMIT_ONE_BIO][SYNC] ino=13009195992709632625, sector=59520, size=16384
[  +0.000013] [BTRFS][MAP_BIO] logical=30474240, len=16384, op=1, devs=2
[  +0.000004] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=38862848, bio_sectors=59520, op=1
[  +0.000004] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=146210816, bio_sectors=59520, op=1
[  +0.000017] [BTRFS][SUBMIT_ONE_BIO][SYNC] ino=13009195992709632625, sector=59584, size=16384
[  +0.000070] [BTRFS][MAP_BIO] logical=30507008, len=16384, op=1, devs=2
[  +0.000003] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=38895616, bio_sectors=59584, op=1
[  +0.000004] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=146243584, bio_sectors=59584, op=1
[  +0.000026] [BTRFS][SUBMIT_EXTENT_PAGE] inode=1 offset=30670848 size=4096 [type=meta] (no dentry)
[  +0.000003] [BTRFS][SUBMIT_ONE_BIO][SYNC] ino=13009195992709632625, sector=59776, size=65536
[  +0.000114] [BTRFS][MAP_BIO] logical=30605312, len=65536, op=1, devs=2
[  +0.000003] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=38993920, bio_sectors=59776, op=1
[  +0.000005] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=146341888, bio_sectors=59776, op=1
[  +0.000009] [BTRFS][SUBMIT_ONE_BIO][SYNC] ino=13009195992709632625, sector=59904, size=16384
[  +0.000012] [BTRFS][MAP_BIO] logical=30670848, len=16384, op=1, devs=2
[  +0.000003] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=39059456, bio_sectors=59904, op=1
[  +0.000003] [BTRFS][STRIPE_BIO] dev=/dev/loop13, devid=1, physical=146407424, bio_sectors=59904, op=1
[ +28.674785] [BTRFS][WRITEPAGES] file=/test inode=257 writeback start
[  +0.000022] [BTRFS][EXTENT_WRITEPAGES] file=/test inode=257 writeback triggered
```

로그는 원본 파일 /test(inode=257)에 대해 writepages가 호출되면서 디스크 쓰기가 시작되는 지점부터 시작된다. 이후 extent_writepage()가 여러 번 호출되며 파일 내용을 페이지 단위로 dirty 처리한다.

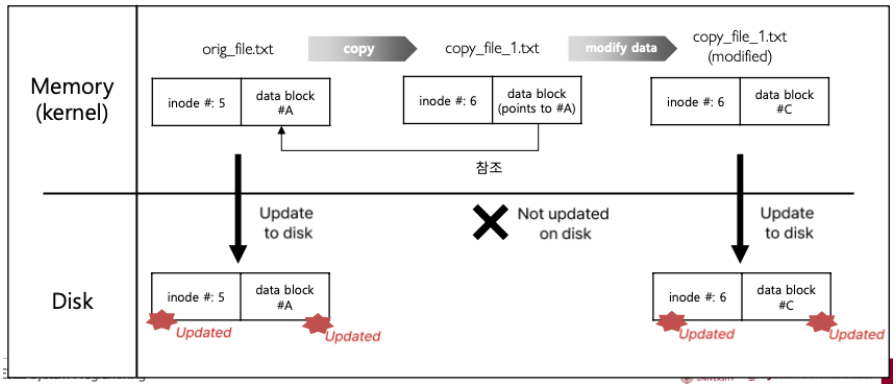
파일 데이터는 submit_extent_page()에 의해 디스크로 flush될 준비가 되며, 그 후 submit_data_bio() → map_bio() → stripe_bio()를 거쳐 디바이스(/dev/loop13)에 실제 bio가 전송된다. 데이터 블록 전송 이후, Btrfs는 내부 메타데이터도 함께 처리하게 되는데, 이 과정에서 submit_extent_page()가 다시 호출되고 type=meta로 로그에 기록된다.

특히 로그에서 submit_one_bio()와 submit_data_bio()가 반복적으로 호출되는 모습은, Btrfs가 데이터뿐만 아니라 내부 구조(메타데이터 트리)까지 지속적으로 업데이트하며 쓰고 있음을 의미한다. map_bio()와 stripe_bio()는 이 데이터를 실제 물리 디바이스로 전송하는 과정이고, 이때 원본 파일에 대한 flush만 이루어지고, 복사본의 경우 디스크로 flush가 되지 않았다는 것을 확인할 수 있다.

실험 분석 및 전제 설명

Btrfs 실험 조건

실험에서의 Btrfs는 CoW(Copy-on-Write) 기반 파일 시스템의 특성을 따른다.



위 첨부한 실습 자료 그림과 같이, 파일 생성 시, 메타데이터와 데이터 블록(data block)은 디스크에 새롭게 쓰이는 반면, 복사할 때는 메타데이터 및 데이터 블록은 디스크에 새롭게 쓰이지 않는다.

본 실험에서는 실제 파일 데이터가 디스크에 쓰였는지를 기준으로 판단하였으며, CoW로 인해 반복적으로 발생하는 메타데이터(write) 로그는 분석 대상에서 제외하였다.

이는 Btrfs가 내부적으로 메타데이터 업데이트를 매우 자주 수행하는 특성이 있지만, 본 실험의 목적은 파일 수정이 아닌, 파일 복사 시 EXT4와의 데이터 쓰기 패턴을 비교하는 데 있으므로, 데이터 write 중심으로 분석한 것이다.

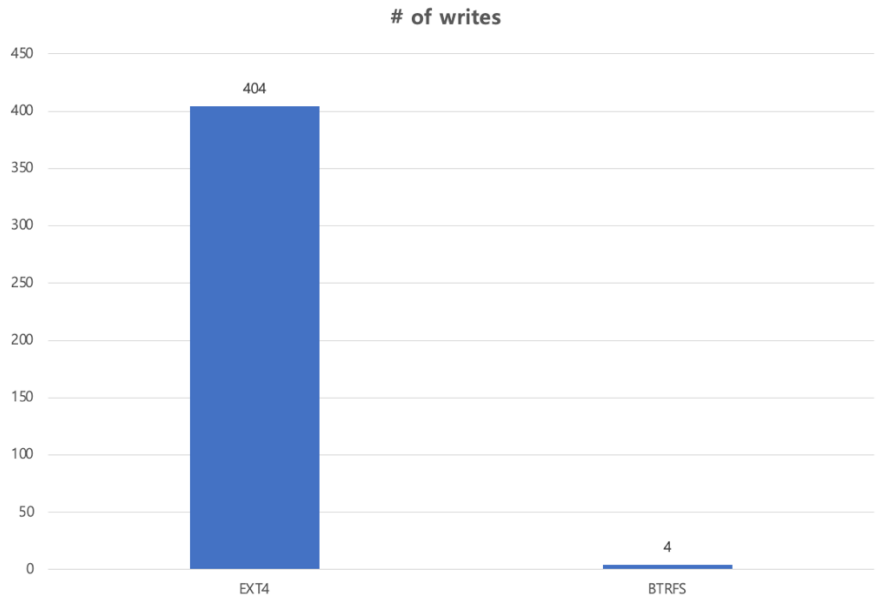
Offset 출력 간격 조정

커널에서 실제 기록되는 로그는 페이지 단위(보통 4KB)로 기록되어, 2MB의 파일을 복사하는데 있 단일 파일 복사만으로도 수많은 로그가 발생할 수 있다.

```
// "test"가 경로에 포함되어 있을 때 && offset이 512KB 단위일 때만 출력
if ((strstr(path, "test") != NULL) && (offset % (1024 * 512) == 0)) {
```

위 사진과 같이, 이를 실험적으로 분석 및 시각화하기 위해 offset 로그는 512KB 간격으로 필터링하여 출력하도록 수정하였고, 이러한 수정은 실험의 본질적인 비교(EXT4 vs Btrfs의 쓰기 패턴 분석)에는 영향을 주지 않는다고 판단하였다.

디스크 쓰기(write) 횟수



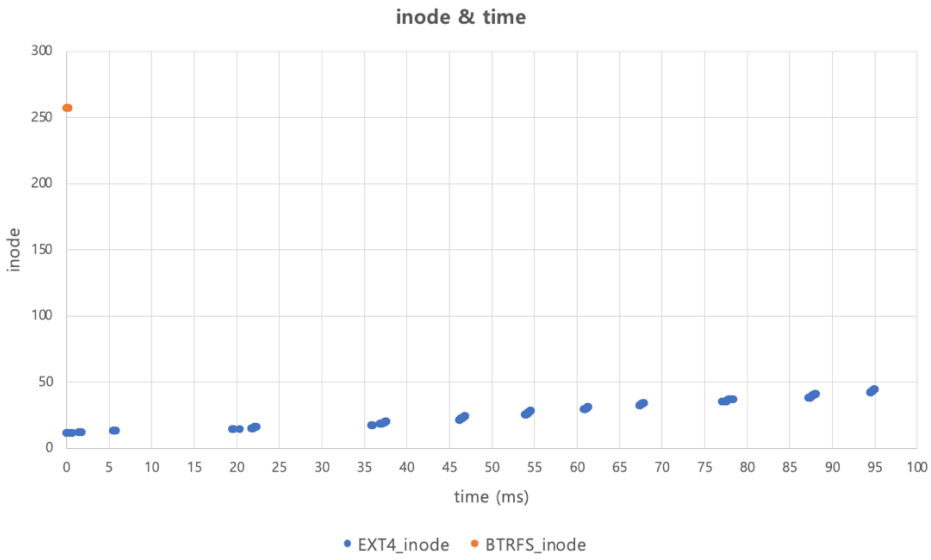
EXT4는 복사된 각 파일에 대해 실제 데이터 블록을 새로 할당하고 디스크에 기록하기 때문에, 100개의 파일 복사 과정에서 총 400개 이상의 실제 데이터 쓰기 I/O가 발생하였다. 이는 EXT4가 데이터 복사 시, 원본 파일과 무관하게 새 블록을 디스크에 할당하고 데이터를 반복적으로 기록한다는 구조적 특성 때문이다.

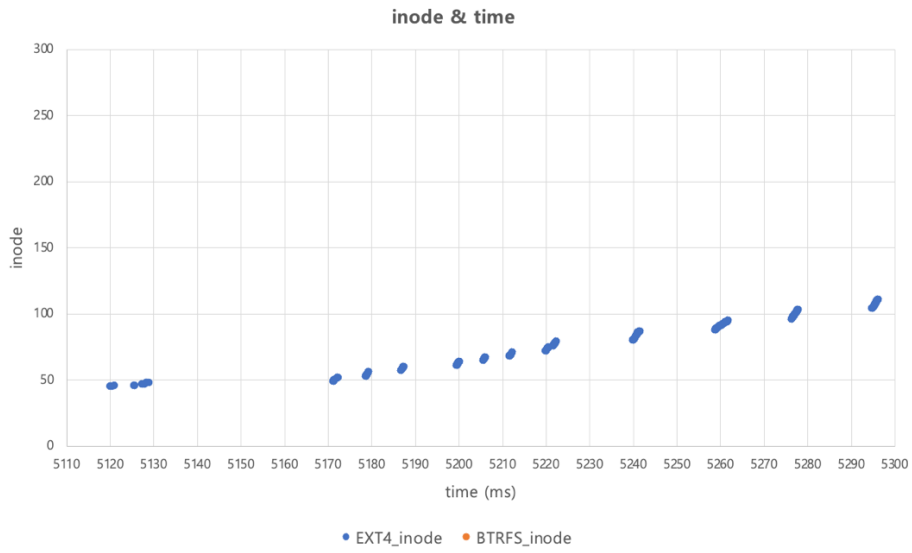
반면, Btrfs는 CoW(Copy-on-Write) 기반의 구조를 가지고 있기 때문에, 파일 복사 시 원본 파일의 데이터 블록을 그대로 참조하고, 새로운 inode만 메모리에 생성한다. 이로 인해 실험 중 복사된 파일이 수정되지 않았기 때문에, 디스크에 실질적인 데이터 쓰기는 거의 발생하지 않았으며, 로그 상으로도 총 4회의 데이터 write만 관측되었다.

그래프에서 나타난 Btrfs의 4회 디스크 쓰기는 복사된 파일에 대한 쓰기가 아니라, 원본 파일을 생성할 때 발생한 데이터 쓰기로 해석된다. 이후 파일 복사 과정에서는 Btrfs의 CoW 특성상 데이터 블록을 복사하지 않고 참조만 하므로, 추가적인 데이터 쓰기는 발생하지 않았다.

따라서, Btrfs는 동일 조건에서 EXT4에 비해 훨씬 적은 디스크 I/O로 파일 복사를 처리한다는 것이 실험을 통해 확인되었다.

inode 업데이트 시점



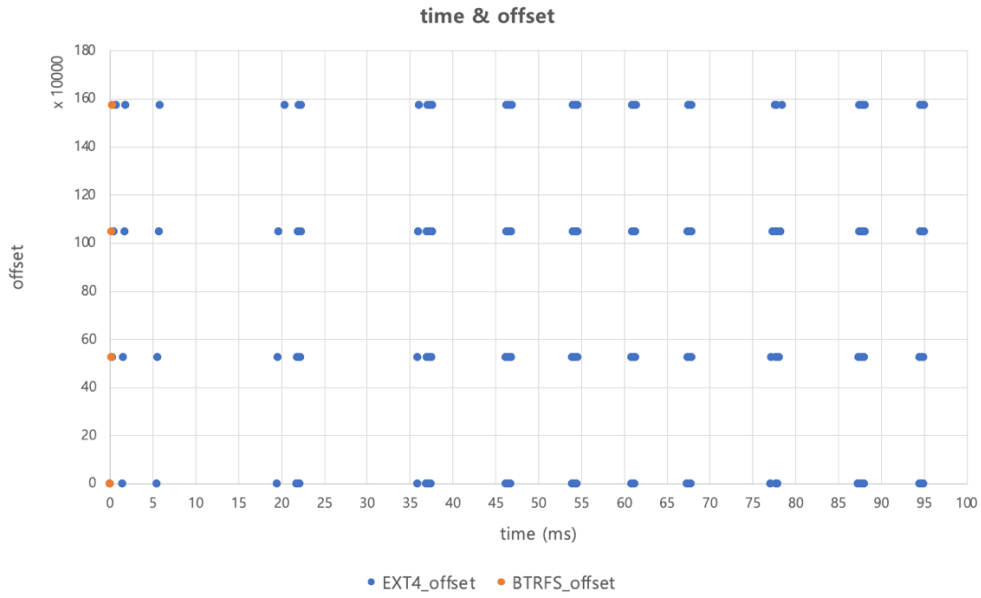


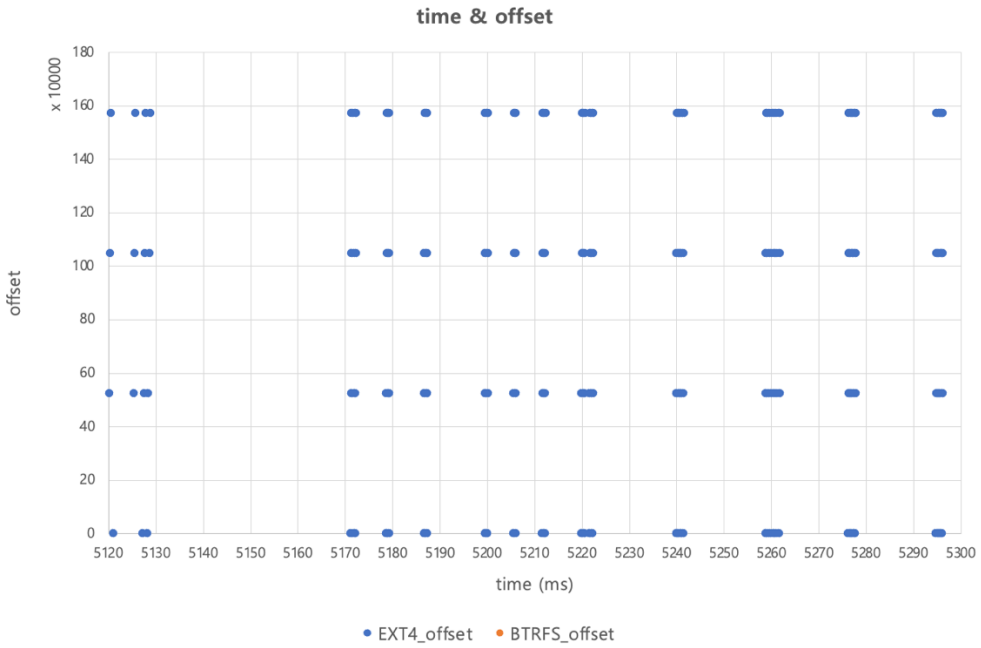
Btrfs의 경우, 복사된 파일의 내용이 수정되지 않으면 실제 디스크에는 아무런 쓰기도 발생하지 않는 CoW(Copy-on-Write) 특성을 갖는다. 이를 반영하듯 그래프 1에서 Btrfs는 단 한 번, 파일이 생성될 때 inode 번호 257이 매우 이른 시점에만 기록되었고 이후 어떤 디스크 업데이트도 발생하지 않았다. 이는 Btrfs가 파일 복사 시 메모리 내에서 inode 및 메타데이터를 생성하지만, 실제 디스크에는 flush하지 않았다는 것을 의미한다.

반면 EXT4는 복사되는 각 파일에 대해 inode가 생성되고, 페이지 단위로 더티 페이지가 디스크에 flush되면서 연속적으로 inode 기록이 발생한다. 그래프 1과 2에서는 시간이 지남에 따라 inode 번호가 순차적으로 증가하는 모습을 볼 수 있으며, 이는 100개의 파일을 복사하는 동안 지속적으로 디스크에 쓰기가 이루어졌음을 보여준다.

결론적으로 Btrfs는 파일 복사 시 디스크에 메타데이터를 반영하지 않아 I/O 횟수가 현저히 적고, EXT4는 모든 복사 파일에 대해 inode 및 데이터가 실제 디스크에 기록되기 때문에 상대적으로 많은 디스크 쓰기가 발생한다. 이러한 차이는 두 파일 시스템의 구조적 차이를 뚜렷하게 보여주는 실험 결과라 할 수 있다.

offset 업데이트 시점





EXT4에서는 파일을 복사할 때 실제 데이터 블록을 직접 디스크에 write하기 때문에 시간에 따라 offset 값이 꾸준히 발생한다. offset은 페이지 단위(4KB)로 쓰이지만 실험에서는 로그를 줄이기 위해 512KB 단위로만 출력하도록 설정했으며, 그럼에도 불구하고 규칙적이고 반복적인 offset write가 발생하고 있는 것이 확인된다. 이는 EXT4가 파일을 복사할 때마다 새로운 데이터 블록을 디스크에 flush하기 때문이다.

BTRFS의 경우 offset 관련 로그는 초기 파일 생성 단계에서만 나타난다. 복사된 100개의 파일은 CoW(Copy-on-Write) 메커니즘에 따라 원본 데이터 블록을 공유하므로, 복사 과정에서는 추가적인 데이터 write가 발생하지 않는다.

어려웠던 점과 해결 방법

저는 이번 실험을 진행하며 여러 가지 어려움이 있었지만, 이를 해결해 나가면서 커널 내부 동작에 대한 깊은 이해를 얻을 수 있었습니다.

가장 먼저 어려웠던 점은 UTM에서의 초기 환경 설정 및 커널 빌드 과정이었습니다. UTM을 이용한 가상 환경 구축 과정에서 Ubuntu 이미지의 디스크 용량 문제와 커널 소스 설정, 컴파일 환경 세팅 등이 까다롭게 느껴졌습니다.

그중에서, 커널 내 파일 시스템 구조와 함수 흐름을 추적하는 과정이 가장 기억에 남습니다. EXT4와 BTRFS는 모두 방대한 구조를 갖고 있어 어떤 함수가 언제 호출되고, 그 내부에서 어떤 처리가 이뤄지는지를 파악하는 데 많은 시간이 들었습니다. 이를 해결하기 위해 VSCode를 설치하여 커널 소스를 구조적으로 탐색하고, 주요 함수 간 흐름을 직접 흐름도 형태로 정리했습니다.

특히 필요한 함수를 찾을 때는

```
grep -rnw './' -e 'btrfs_submit_metadata_bio'
```

와 같은 명령어를 통해 전체 커널 디렉토리에서 호출 지점을 찾아내고, 관련 흐름을 하나씩 추적해나갔습니다.

가장 중요한 해결 방법은 직접 printk()를 활용해 로그를 삽입하며 확인해보는 방식이었습니다. 파일을 생성하거나 복사하는 시점에 어떤 함수들이 호출되는지, 어떤 inode나 offset이 디스크에 기록되는지를 실험적으로 로그를 통해 검증해보며 실질적인 흐름을 파악할 수 있었습니다.

이러한 일련의 과정을 통해 단순히 이론적으로만 이해하던 파일 시스템 내부의 동작을, 실제 커널 수준에서의 코드 실행 흐름과 연계해 구체적으로 파악할 수 있었던 것 같습니다.