시스템프로그래밍 과제 1

2021320097 이현서

2025.4.14(월)

1. 배경지식설명

1-1. Copy-on-Write (CoW)

Cow 는 파일을 복사하여 새로운 파일을 생성할 때, 원본 파일 전체를 모두 새롭게 디스크에 쓰는 것이 아니라 resource 를 공유하다가 원본 파일간의 차이가 생기는 시점에 해당 차이를 디스크의 새로운 위치에 저장하는 방법이다.

파일을 복사하여도 메모리 상에서 inode 는 새로 생성되지만 data block 은 원본 파일을 참조하고 있고, 수정이 일어나지 않는 이상 디스크로의 새로운 업데이트는 일어나지 않는다. 그 후 수정이 일어나게 메모리 내 새로운 data block 을 할당하게 되고 변경된 data block 과 새로운 inode 가 디스크에 쓰인다.

CoW 를 쓰지 않을 때와 다르게 불필요한 메모리 복사와 자원 낭비를 줄일 수 있다. 이로 인해 CoW 는 ZFS, Btrfs, ReFS, Bcachefs 등 현대적인 파일시스템의 핵심 메커니즘으로 채택되고 있다.

1-2. Btrfs

Btrfs 는 B-tree-fielsystesm 의 약자로 리눅스의 파일 시스템 중 하나로 이름에서 볼 수 있듯이 B-tree 를 기반으로 데이터 밸런싱을 한다. 또한 위에서 말한 Copy-on-Write 을 채택한 파일 시스템이기도 하다. 스냅샷을 지원하여 백업에 용이하다는 점과 SSD 환경에 최적화 되어 있다는 특징 등을 갖고 있다.

1-3. Ext4

Ext4 는 리눅스의 확장 파일 시스템(Ext)의 4 번째 버전으로 ext 1,2,3 과도 역호환이 가능하다. 대표적인 특징으로는 journaling 을 지원한다는 특징을 갖고 있어 시스템 crash 와 같은 상황에서 복구가 용이하다. 또한 block 단위가 아니라 extent 단위로 공간을 할당하고 데이터를 disk 로 쓰기전 까지 최대한 버퍼에 모아두는 delayed allocation 방법을 채택하고 있다. 하지만 CoW 와 스냅샷은 지원하지 않는다.

2. 작성한 소스코드에 대한 상세한 설명

2-1. Ext4 (linux-5.15/fs/ext4/page-io.c)

```
void ext4_io_submit(struct ext4_io_submit *io)
```

이 함수는 ext4 파일시스템에서 쓰기 요청이 들어왔을 때 실제로 block io 를 제출하는 함수로 메모리에서부터 디스크로 쓰여지는 시점에 실행되는 함수라고 할 수 있다. 이 함수 내의 코드를 추가하여 이 시점에서의 path, inode, offset 로그를 출력하였다.

```
struct ext4_io_submit {
	struct writeback_control *io_wbc;
	struct bio		*io_bio;
	ext4_io_end_t		*io_end;
	sector_t		io_next_block;
};
```

struct ext4_io_submit 은 ext4.h 에 위와 같이 선언되어 있는 것을 확인하였고

```
struct inode *inode = io->io_end->inode;
```

bio 가 속한 파일의 inode 정보를 가져올 수 있다.

```
struct dentry *dentry;
char pathname[256] = "unknown";
char *tmp_path = pathname;
```

경로 정보를 저장할 문자열 변수를 선언하고 기본값을 unknown 으로 설정하였다.

```
dentry = d_find_alias(inode);
		if (dentry) {
			tmp_path = dentry_path_raw(dentry, pathname, sizeof(pathname));
			if (IS_ERR(tmp_path)) {
				tmp_path = pathname;
			}
		}
```

inode 로부터 dentry 를 찾고(d_find_alias() 함수 사용) 해당 파일의 경로를 문자열로 갖고 온다(dentry_path_raw() 함수 사용). 실패할 경우는 기본값을 사용한다.

```
loff_t offset = (loff_t)bio->bi_iter.bi_sector << 9;
```

bio 가 쓰는 디스크의 실제 offset 을 byte 단위로 계산한다.(512byte 단위를 9bit shift 하여 byte 단위로 변환하였음)

```
if (strstr(tmp_path, "/test") != NULL) {
			printk(KERN_INFO "[EXT4-SUBMIT] path=%s inode=%lu offset=%llu₩n",
				tmp_path,
				inode->i_ino,
				(unsigned long long)offset);
		}
```

경로가 test 디렉토리일 경우만 경로, inode, offset 을 로그로 출력하였다.

```
if (dentry)
                dput(dentry);
```

dentry 사용 후 참조 해제를 통해 리소스를 정리하였다.

2-2. Btrfs (linux-5.15/fs/btrfs/extent-io.c)

```
static int submit_extent_page(unsigned int opf,
                              struct writeback_control *wbc,
                              struct btrfs_bio_ctrl *bio_ctrl,
                              struct page *page, u64 disk_bytenr,
                              size_t size, unsigned long pg_offset,
                              bio_end_io_t end_io_func,
                              int mirror_num,
                              unsigned long bio_flags,
                              bool force_bio_submit)
```

이 함수도 마찬가지로 btrfs 파일 시스템에서 bio 를 디스크에 제출하기 직전에 호출되는 함수이다.

```
struct btrfs_inode *inode = BTRFS_I(page->mapping->host);
```

btrfs 기존 코드에서는 위 코드처럼 페이지가 속한 파일의 VFS inode 를 btrfs 전용 구조체인 btrfs_inode 로 변환하고 있다.

```
struct inode *vfs_inode = &inode->vfs_inode;
```

d_find_alias(), dentry_path_raw 와 같은 VFS 관련 함수들을 사용하기 위해 btrfs 구조체에서 VFS inode 를 접근하는 코드를 추가하여 inode 를 가져왔다.

```
struct dentry *dentry;
char pathname[256] = "unknown";
char *tmp_path = pathname;
dentry = d_find_alias(vfs_inode);
                if (dentry) {
                tmp_path = dentry_path_raw(dentry, pathname, sizeof(pathname));
                if (IS_ERR(tmp_path))
                        tmp_path = pathname;
                }
```

이 후 경로에 대한 과정은 ext4 와 같은 방식으로 코드를 구성하였다.

```
u64 physical_offset = disk_bytenr + offset;
```

여기서는 bio 에서 따로 offset 을 가져오지 못하기에 매핑 시작 byte 주소에 현재 page 내 offset 을 더하여 실제 byte offset 을 계산하였다.

```
if (strstr(tmp_path, "/test") != NULL) {
                printk(KERN_INFO "[BTRFS-SUBMIT] path=%s inode=%lu offset=%llu\n",
```

```
                        tmp_path,
                        vfs_inode->i_ino,
                        (unsigned long long)physical_offset);
                }
if (dentry)
                dput(dentry);
```

ext4 와 마찬가지로 경로가 test 디렉토리일 경우만 경로, inode, offset 을 로그로 출력하고 dentry 참조 해제를 통해 리소스를 정리하였다.

3. 테스트 환경구동에 대한 간략한 설명

3-1 가상 디스크 생성 및 마운트에 대한 간략한 설명

```
#!/bin/bash
sudo losetup /dev/loop13 /home/alex4242/sp_proj01/ext4dfile
sudo mount -t ext4 /dev/loop13 /home/alex4242/sp_proj01/ext4_dir

sudo losetup /dev/loop14 /home/alex4242/sp_proj01/btrfsdfile
sudo mount -t btrfs /dev/loop14 /home/alex4242/sp_proj01/btrfs_dir
```

ext4dfile 과 btrfsdfile 을 각각 loop13,14 디바이스에 연결하여 가상 디스크를 생성하였다. 그 후 연결된 가상디스크를 각각 ext4, btrfs 파일시스템으로 ext4_dir, btrfs_dir 에 마운트하였다. 이 과정을 mount_disks.sh 스크립트로 만들어 재부팅 할 때마다 이 과정을 실행시켜주었다.

3-2 테스트 스크립트에 대한 간략한 설명

```
#!/bin/bash
home_dir=/home/alex4242/sp_proj01/ext4_dir
sudo rm -rf ${home_dir}/*
sudo dd if=/dev/zero of=${home_dir}/test bs=1024 count=2000
for i in {1..100}; do
   sudo cp ${home_dir}/test ${home_dir}/test_$i
done
#!/bin/bash
home_dir=/home/alex4242/sp_proj01/btrfs_dir
sudo rm -rf ${home_dir}/*
sudo dd if=/dev/zero of=${home_dir}/test bs=1024 count=2000
for i in {1..100}; do
   sudo cp --reflink=auto ${home_dir}/test ${home_dir}/test_$i
done
```

실험 대상 디렉토리 경로를 home_dir 에 저장하고 기존 모든 파일을 삭제하여 이전 실험 결과물을 제거하였다. 그 후 2MB 파일을 생성하여 cp 명령어를 통해 test_1 부터 test_100 까지 복사하였다. 이 과정을 testscript_ext4 와 testscript_btrfs.sh 테스트 스크립트로 작성하였다.

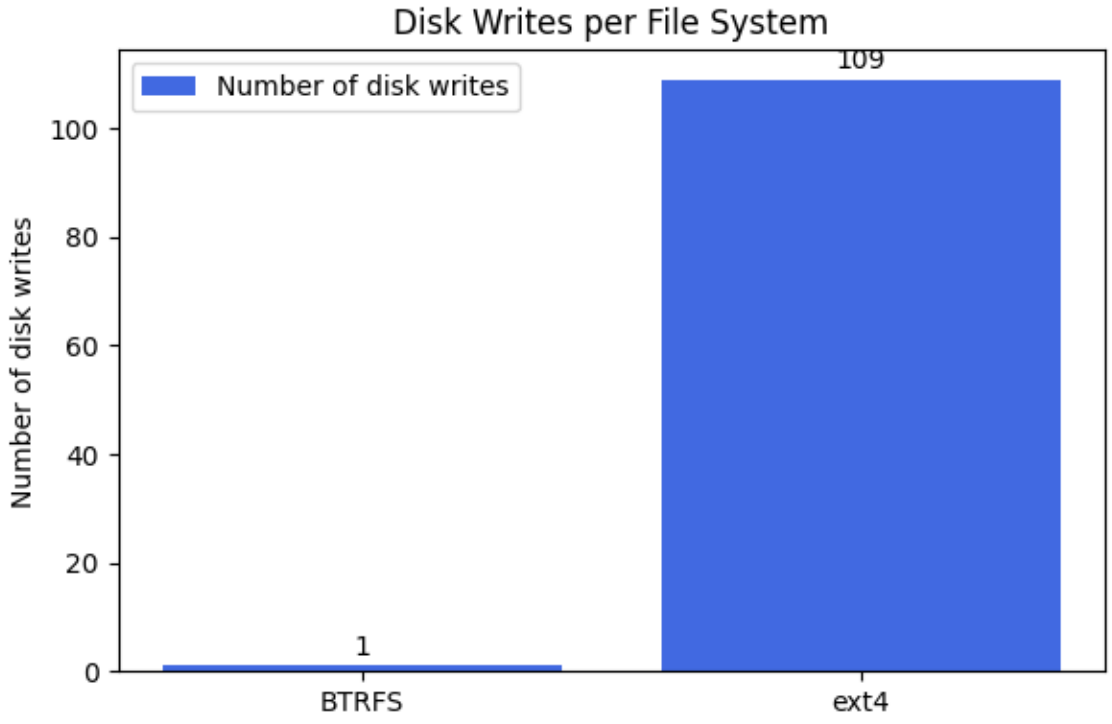
4. 실험결과에 대한 비교분석

본 실험은 동일한 테스트 스크립트를 EXT4 와 BTRFS 파일 시스템에 각각 실행하여, 디스크 쓰기 횟수, inode 업데이트 패턴, offset 변화 양상을 비교하였다. 이를 통해 전통적인 파일 시스템(EXT4)과 Copy-on-Write 방식의 파일 시스템(BTRFS) 간의 내부 동작 차이를 확인할 수 있었다. 아래의 모든 그래프는 dmesg 로그를 추출하여 파이썬 matplotlib 를 통해 그렸다.

4-1. 디스크 쓰기 횟수 비교

EXT4 는 총 109 회의 디스크 쓰기가 발생한 반면, BTRFS 는 단 1 회의 디스크 쓰기만 발생하였다. 이는 두 파일 시스템의 설계 차이에서 비롯된다. EXT4 는 파일 생성 및 복사 시 매번 inode 및 데이터 블록을 갱신하며 디스크에 직접 반영하지만, BTRFS 는 Copy-on-Write 특성에 따라 처음 파일 생성 시에만 디스크 쓰기가 일어나고 수정이 되지 않는 한 추가적인 디스크 쓰기가 일어나지 않는다.

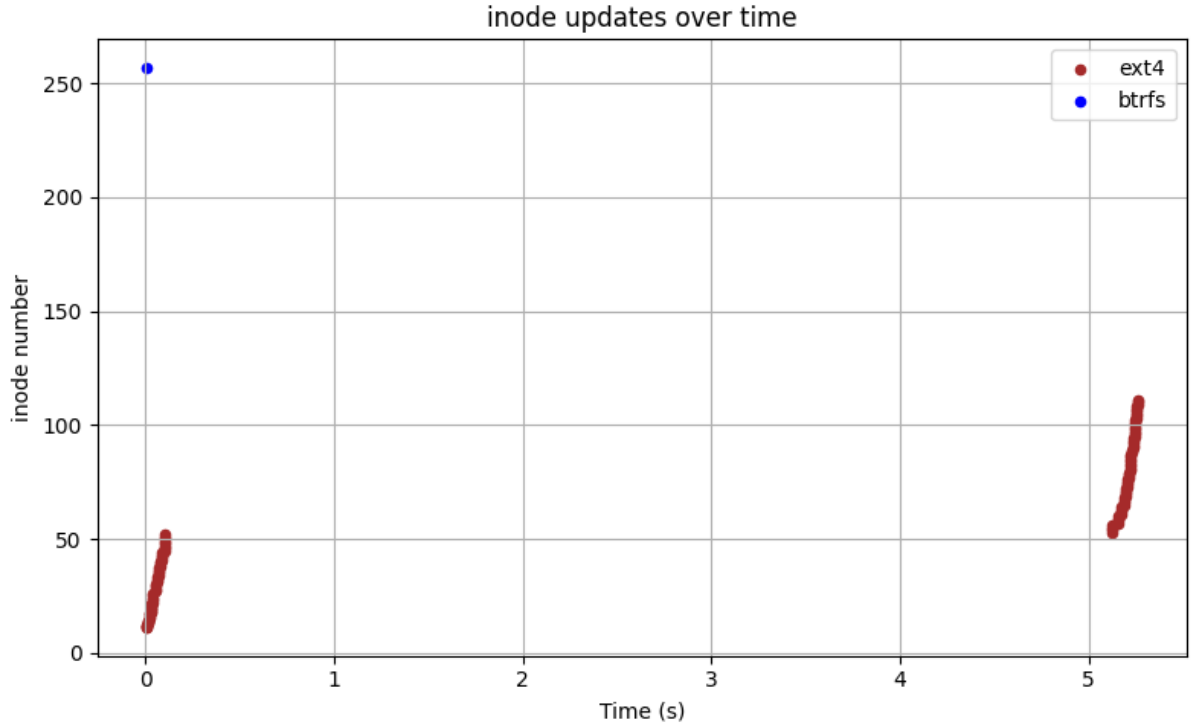
EXT 4 에서는 원본 파일을 비롯한 초반 복사 파일들에 대해 1MB 씩 두 번 나눠 쓰기가 발생하여 예상하였던 101 회의 디스크 쓰기보다 많은 횟수가 발생하였다. 이는 초기 구간 여러 조건때문에 한 번에 블록을 담아 쓰기를 못한 것으로 예상된다.



4-2. inode 업데이트 시점 분석

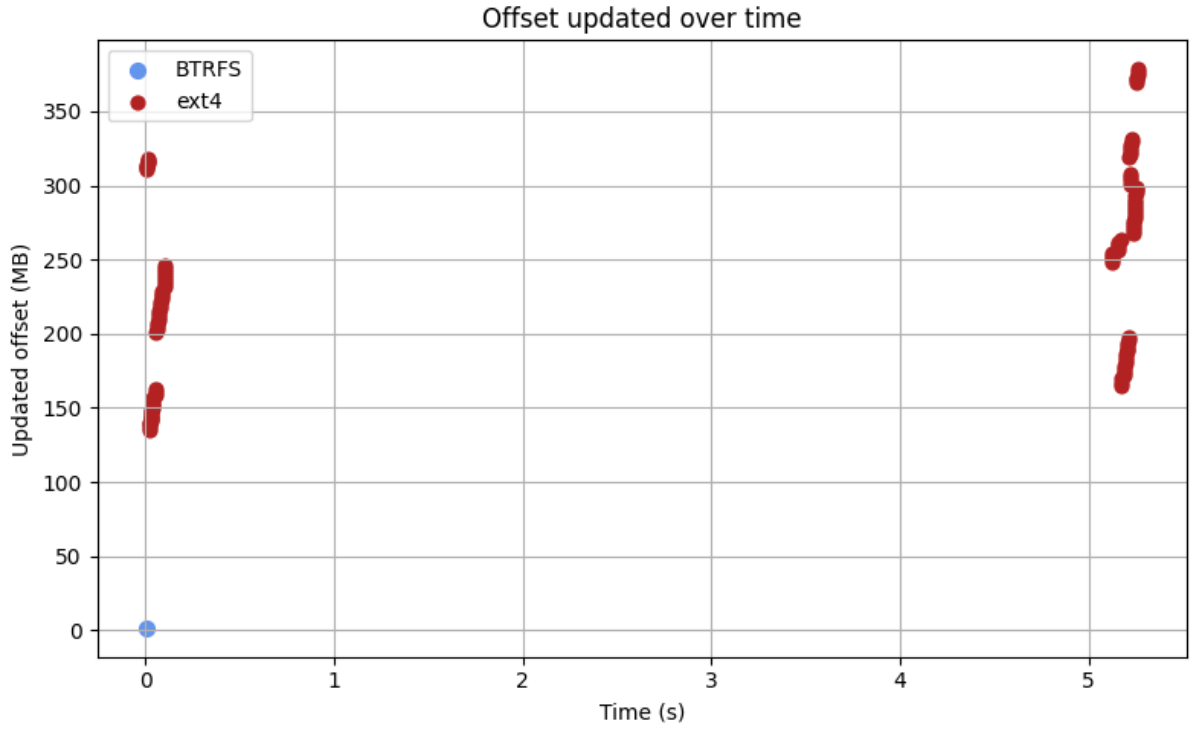
EXT4 에서는 복사되는 각 파일마다 inode 가 새로 할당되고, 매번 디스크에 기록됨을 확인할 수 있다. 이로 인해 시간 축을 따라 11 부터 지속적으로 증가하는 inode 번호가 나타나며, BTRFS 에서는 단 한 번의 inode 업데이트 로그만 확인된다. 이는 BTRFS 의 CoW 특성상 실질적인 변경이 발생할 때에만 inode 를 갱신함을 의미한다.

또한 EXT4 로그에서는 약 5 초 정도의 공백이 발생하는데, 이는 ext4 의 delalyed allocation 혹은 리눅스 커널 내부적인 정책에 의해, 일정량의 쓰기가 완료된 후 내부적으로 캐시를 flush 하면서 디스크 접근이 일시적으로 멈춘 것으로 예측된다. 실험 당시 41 개의 파일까지는 빠르게 기록되었지만, 로그 출력이 지연되었고, 재개 시점이 5 초 후로 기록되었다.



4-3. offset 변화 분석

EXT4 에서는 offset 값이 꾸준히 증가하지만, 일정한 간격으로 점프하며 변화하는 특징을 보인다. 이는 파일이 디스크에 저장될 때 연속적인 블록을 우선 배치하지만, 내부적으로는 블록 정렬, 단편화 등의 영향으로 연속되지 않은 블록에 저장되는 현상으로 예측된다. BTRFS 는 위에서 말했듯이 offset 또한 한 번만 기록된다.



4-4. 결론

이번 실험을 통해 EXT4 와 BTRFS 파일 시스템의 구조적 차이가 실질적인 디스크 동작 방식에 미치는 영향을 확인할 수 있었다. BTRFS 는 CoW 기반 설계로 인해 디스크 쓰기 횟수를 획기적으로 줄이며 본 실험과 같은 환경에서는 높은 성능을 보여줄 수 있다는 것을 확인하였다.

5. 과제수행시 어려웠던 부분과 자신의 해결방법

;

5-1. jouranl log 디스크 저장

journal log 설정에서 storage=auto 로 되어있어, 테스트 할 때마다 디스크에 무한으로 로그가 쌓여 쓸데없는 용량을 크게 차지하고 있었다. storage=None 으로 설정하여 디스크 저장을 막아서 해결하였다.

5-2. 로그 묻힘 현상

테스트 할 때 write 동작을 할 때마다 printk()가 실행되기 때문에 파일 복사 관련 동작이 아닌 다른 동작 때문에 상당히 많은 로그가 떠서 파일 복사 관련 로그들이 묻혀 파일 복사 관련 로그가 안생기는 걸로 잘 못 인지하게 되었다.. (로그들이 disk 에 쓰이고 있었기 때문에 이에 따른 수 많은 로그들이 생긴걸로 예측됨) printk() 를 "file"이 경로에 있을 때만 실행하는 것으로 수정하여 해결하였다.

5-3. loop device

loop 10 까지 사용되고 있어 loop 11,12 에 마운트하였는데, 테스트 후 부팅을 반복하는 과정 중에 loop 11,12 이 다른 것에 의해 할당되었다. 명확한 이유는 차지 못했지만 마운트 스크립트를 수정하여 loop 13,14 를 사용하였다.

5-4. 수정해야 할 함수 찾기

어느 레벨의 함수까지 들어가서 로그를 찍어야하는지에 대해 고민이 많이 되었고, 다양한 함수들에 로그를 찍어보았다. 커널이 다시 빌드하는데 10 분이 넘게 걸려 여러 함수를 테스트하는 과정 중에 물리적으로 시간이 많이 들었다.(빌드 속도를 올려보자 했지만 마땅한 방법은 찾지 못함.) 결과적으로 로그가 괜찮게 뜨고 최대한 메모리에서 디스크로 쓰는 시점에 가까운 함수를 선정하였다.