Operating System, Section 2

Assignment4 - Simple File System Document

Yoo, J. H.

32212808

Department of Mobile Systems Engineering

2025-12-01

Table of Contents

1. 서론

A. 개발 목적

본 프로젝트의 목적은 운영체제의 파일 시스템을 이해하고, 프로그래밍을 통해 직접 구현하는 데 있다.

구체적으로 동적으로 생성, 편집될 수 있는 파일을 비휘발성 저장 장치에 저장하고 관리하는 파일 시스템을 구현하고, 사용자가 파일을 쉽게 찾을 수 있도록 하기 위한 논리적인 파일 위치를 비휘발성 저장 장치의 물리적 위치로 매핑하는 기능을 구현함으로써, 단순히 파일 시스템을 이해하는 것에서 나아가 실질적인 운영체제의 동작 과정을 이해하는 것을 목표로 한다.

또한 Index Node(i-node) table 의 위치와 개수 등의 메타데이터를 관리하는 자료구조인 Superblock 과 파일의 실제 내용이 저장되는 데이터 블록을 사용해 동작하는 파일 시스템을 단순하게 구현함으로써 파일 시스템의 전반적인 동작과 운영체제를 종합적으로 학습하는 것이 본 프로젝트의 핵심이다.

2. 소프트웨어 요구사항 기술 (Requirements)

A. 기능적 요구사항

Makefile 작성
프로그램을 make 명령어로 컴파일할 수 있어야 한다.
mount
disk.img 의 superblock 과 i-node table 을 읽어 file system 내부 memory 구조에 연결할 수 있어야 한다.
디스크 이미지 생성 도구
운영체제가 파일 시스템을 mount 하기 전에 비휘발성 저장 장치를 미리 생성해야 한다. 파일 시스템을 초기화한 후 superblock 정보, i-node 테이블 정보, 생성된 모든 파일 목록을 출력해야 한다.
open
File 이름으로 directory 를 탐색하여 해당 file 의 i-node 를 찾는 기능을 제공해야 한다.
디렉토리 entry 캐시 & 해시
File 이름으로 directory 를 탐색하는 경우, 자주 접근하는 directory entry 를 cache 에 저장하고, hash 구조를 사용하여 disk 접근 없이 i-node 번호를 빠르게 찾을 수 있어야 한다.
read
File 의 i-node 정보를 이용하여 해당 data block 을 찾아 내용을 불러오는 기능을 제공해야 한다.
write
File 이름을 기준으로 file 을 생성하거나, 기존 file 의 내용을 덮어쓰고 i-node 의 정보를 갱신하는 기능을 제공해야 한다.
버퍼 캐시
File 의 data 를 읽거나 쓰는 경우, buffer cache 를 사용하여 disk.img 접근 횟수를 줄여야 한다.
close
File 연산이 종료된 경우, 사용한 내부 구조체를 정리해야 한다.
출력
각 File 연산의 수행 내용과 결과를 출력해야 한다.

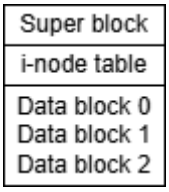
B. 비기능적 요구사항

성능
인자를 받아 디렉토리 entry 캐시 & 해시 사용 여부와 버퍼 캐시 사용 여부를 받고, 아래 4 가지 경우에서의 실행 시간을 제시해야 한다. 디렉토리 entry 캐시 & 해시 미사용 + 버퍼 캐시 미사용 디렉토리 entry 캐시 & 해시 미사용 + 버퍼 캐시 사용 디렉토리 entry 캐시 & 해시 사용 + 버퍼 캐시 미사용 디렉토리 entry 캐시 & 해시 사용 + 버퍼 캐시 사용
안정성
본 프로젝트는 단일 process 환경을 가정하기 때문에 동시성 제어, Deadlock 문제, starvation 문제를 고려하지 않는다.
에러 처리
입력된 인자가 잘못된 경우, 오류 메시지를 출력해야 한다.
유지보수성
기능을 각각의 함수로 분리해야 한다.

3. 소프트웨어 설계 (Design)

A. 아키텍처 설계 - file system 전체 구조

본 프로젝트의 file system 은 아래 그림과 같이 단일 partition 구조를 가지며, super block, i-node table, data block 영역으로 구성된다. Mount 할 경우, disk image 를 memory 에 partition 구조체로 불러와서 모든 file 연산을 memory 기반으로 수행한다.



B. 아키텍처 설계 - 주요 자료구조 설계

구조체	역할
super_block	File System metadata
inode	File 의 크기, type, data block 의 위치
dentry	Directory entry
dentry_cache	File 이름을 통해 i-node 번호를 caching
buffer cache	Block 번호를 통해 data 를 caching

C. 아키텍처 설계 - file 접근 처리 흐름

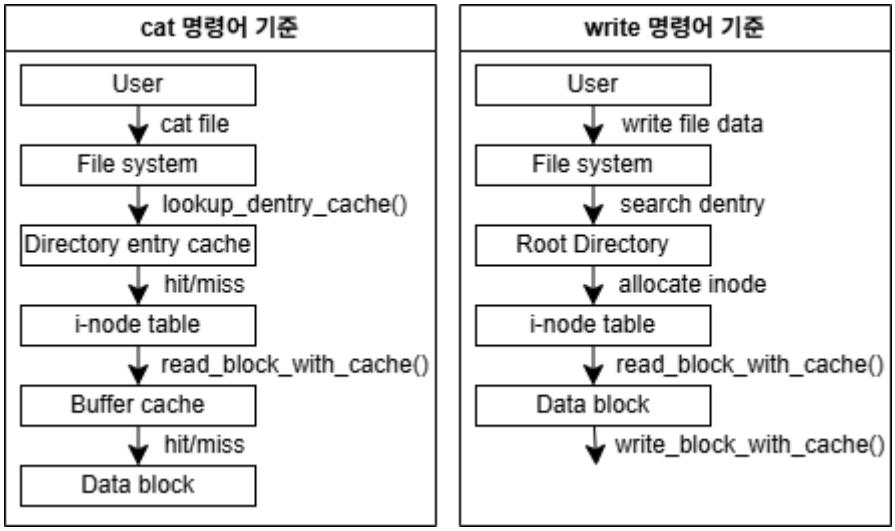
Cat 명령어를 기준으로 아래 표와 같은 흐름으로 동작한다.

1	사용자가 cat 명령어를 입력한다.
2	dentry cache 에서 file 의 이름을 조회한다.
3	cache miss 인 경우 directory block 을 탐색한다.
4	i-node 번호를 얻고 i-node table 을 참조한다.
5	i-node 에 기록된 data block 을 순차적으로 읽는다.
6	buffer cache 를 통해 disk 접근 횟수를 줄인다.

D. 아키텍처 설계 - cache 정책과 설계

directory entry cache 는 file 이름 기반 탐색 시간을 줄이기 위해 사용한다. buffer cache 는 동일한 data block 에 반복 접근할 경우 disk 접근 횟수를 줄이기 위해 First Input First Output(FIFO) 교체 정책으로 구현한다.

E. 아키텍처 설계 - 시퀀스 다이어그램



4. 소프트웨어 구현 (Implementation)

A. 개발 환경 & 도구

Operating System	
Ubuntu 22.04.5 LTS	
Kernel	
Linux 5.15.0-136-generic	
IDE (Integrated Development Environment) & Editor	
vi	
Programming language	
C	
Framework	
stdio.h	Standard input & output library
stdlib.h	Standard library
string.h	String library
time.h	Time library
unistd.h	POSIX API
_POSIX_C_SOURCE 200809L	POSIX.1-2008 표준 기능을 활성화하여 clock_gettime() 등 POSIX API 를 사용하기 위해 설정
Build tool	
GCC compiler with Makefile	

B. 모듈별 기능 설명

fs.h
File system 에 사용되는 상수, 자료구조를 포함하는 header file 이다.
main.c
File system 의 주요 기능을 구현한 module 로, disk image 생성, mount, file I/O, cache 관리, 사용자 명령어 interface 등을 담당하는 c file 이다.

C. 주요 알고리즘 설명 & Code Snippet

```c
// Bitmap을 사용하지 않고, 새로운 data block을 순차적으로 할당하는 함수
int allocate_data_block(void) {
    static int next_free = 1;
    if (next_free < MAX_BLOCKS) {
        int phys = disk.s.first_data_block + next_free;
        next_free++;
        return phys;
    }
    return 0;
}
```

본 프로젝트는 구현의 단순화를 위해 bitmap 을 사용하지 않고, data block 을 순차적으로 할당하는 방식을 사용하였다.

```c
// Data를 씀
struct inode *file_inode = &disk.inode_table[inode_idx];
size_t len = strlen(content);
int required_blocks = (len + BLOCK_SIZE - 1) / BLOCK_SIZE;
int written = 0;
for (int i = 0; i < required_blocks && i < 6; i++) {
    if (file_inode->blocks[i] == 0) {
        file_inode->blocks[i] = allocate_data_block();
        if (file_inode->blocks[i] == 0) {
            printf("· Writing stopped because can not allocate data block\n");
            break;
        }
    }
    unsigned char buf[BLOCK_SIZE];
    memset(buf, 0, BLOCK_SIZE);
    int chunk = (len - written > BLOCK_SIZE) ? BLOCK_SIZE : (len - written);
    memcpy(buf, content + written, chunk);
    write_block_with_cache(file_inode->blocks[i], buf);
    written += chunk;
}
file_inode->size = written;
printf("· %d bytes is written in %s\n", written, filename);
```

File 의 data 는 i-node 에 기록된 data block pointer 를 따라 순차적으로 저장되며, 쓰기 완료 후 file 의 크기 정보를 i-node 에 갱신한다.

```c
// Directory entry cache 에서 indexing 을 하기 위한 문자열 hash 함수
unsigned int hash_func(const char *str) {
    unsigned int hash = 5381;
    int c;
    while ((c = *str++)) hash = ((hash << 5) + hash) + c;
    return hash % DENTRY_HASH_SIZE;
}

// File 의 이름을 통해 i-node 번호 cache 를 조회하는 함수
int lookup_dentry_cache(const char *name) {
    if (!g_dentry_cache_enabled) return -1;
    unsigned int idx = hash_func(name);
    if (dentry_cache[idx].valid &&
strcmp(dentry_cache[idx].name, name) == 0) {
        g_cache_hits++;
        return dentry_cache[idx].inode_num;
    }
    g_cache_misses++;
    return -1;
}
```

File 이름을 빠르게 탐색하기 위해 문자열 hash 기반의 directory entry cache 를 구현하였다. Cache 가 활성화된 경우 hash 값을 통해 즉시 i-node 번호를 조회하며, cache miss 인 경우, directory block 을 직접 탐색한다.

```c
// Block을 읽는 함수
int read_block_with_cache(unsigned int block_num, void *buf) {
    // Buffer cache를 탐색
    if (g_buffer_cache_enabled) {
        for (int i = 0; i < BUFFER_CACHE_SIZE; i++) {
            // Hit인 경우
            if (buffer_cache[i].valid &&
buffer_cache[i].block_num == block_num) {
                memcpy(buf, buffer_cache[i].data, BLOCK_SIZE);
                g_cache_hits++;
                return 0;
            }
        }
    }
    g_cache_misses++;

    // Miss인 경우 memory에 불러온 disk 구조체에서 block을 읽음
    unsigned char *ptr = get_block_ptr(block_num);
    if (!ptr) return -1;
    memcpy(buf, ptr, BLOCK_SIZE);

    // FIFO 방식으로 cache를 update
    if (g_buffer_cache_enabled) {
        buffer_cache[buffer_head].block_num = block_num;
        memcpy(buffer_cache[buffer_head].data, buf,
BLOCK_SIZE);
        buffer_cache[buffer_head].valid = 1;
        buffer_head = (buffer_head + 1) % BUFFER_CACHE_SIZE;
    }
    return 0;
}
```

Buffer cache 는 최근 접근한 data block 을 memory 에 저장하여 반복적인 disk 접근을 줄이기 위해 사용하였다. Cache 가 가득 찬 경우, FIFO 방식으로 가장 오래된 block 을 교체한다.

```
// Directory entry cache 사용
int existing_inode = lookup_dentry_cache(filename);
if (existing_inode != -1) {
    inode_idx = existing_inode - 1;
} else {
    for(int i=0; i<6; i++) {
        if(root->blocks[i]==0) continue;
        unsigned char buf[BLOCK_SIZE];
        read_block_with_cache(root->blocks[i], buf);
        int offset=0;
        while(offset < BLOCK_SIZE) {
            struct dentry *d = (struct dentry*)(buf+offset);
            if(d->inode == 0) break;
            if(strcmp((char*)d->name, filename)==0) {
                existing_inode = d->inode;
                insert_dentry_cache(filename, existing_inode);
                break;
            }
            offset += d->dir_length;
        }
        if(existing_inode != -1) break;
    }
    if (existing_inode != -1) {
        inode_idx = existing_inode - 1;
    }
}
```

Directory entry 는 block 내부를 순차적으로 탐색하여 file 이름과 일치하는 entry 를 찾는다.

5. 시험 & 디버깅 (Verification)

시험 목록	입력	출력	상태
Makefile 빌드	make	.o 파일 생성	Pass
Makefile 클리어	make clean	.o, .img 파일 삭제	Pass
Program 실행	./main	Command shell 실행	Pass
mkfs 인자 없음	mkfs	인자 입력 대기	Pass
mkfs 인자 있음	mkfs a.img	File System maked(a.img)	Pass
mount 인자 없음	mount	인자 입력 대기	Pass
mount 잘못된 인자	mount b.img	File system mount failed No such file or directory	Pass
mount 정상 인자	mount a.img	File system Mounted(a.img, Root Inode Index: 2)	Pass
write 인자 없음	write	인자 입력 대기	Pass
	write c.txt	인자 입력 대기	Pass
write 인자 있음	write c.txt hello	File created(c.txt, i-node 1) 5 bytes is written in c.txt write time: 45453 ns	Pass
cat 인자 없음	cat	인자 입력 대기	Pass
cat 잘못된 인자	cat d.txt	File is not exist cat time: 21844 ns	Pass
cat 정상 인자	cat c.txt	File 의 data 출력, cat time: 33034 ns	Pass
ls	ls	Directory 내부 요소 출력	Pass
set_cache 인자 없음	set_cache	인자 입력 대기	Pass
	set_cache dentry	인자 입력 대기	Pass
set_cache 잘못된 인자	set_cache wrong wrong	Usage: set_cache [dentry|buffer] [on|off]	Pass
set_cache 정상 인자	set_cache dentry on	Dentry cache is on	Pass
cache_stat	cache_stat	Cache 의 상태, hit, miss 출력	Pass
인자 과잉	mkfs a.img extra	실행 후 error message 출력 Unknown command	Pass
	mount a.img extra		Pass
	write c.text hello extra		Pass
	cat c.txt extra		Pass
	ls extra		Pass
	set_cache dentry on extra		Pass
	cache_stat extra		Pass
exit	exit	Command shell 종료	Pass

아래 표와 같이 비기능적 요구사항의 성능에서 요구하는 cache 성능 측정을 시험하였다. 수행 명령은 cat file 로 통일하였다.

Directory entry cache	Buffer cache	실행 시간(ns)	순위
OFF	OFF	34440	4
	ON	31160	3
ON	OFF	29620	2
	ON	29343	1

시험 결과 Buffer cache 만 사용하는 경우, data I/O 비용만 감소하게 되며, buffer cache 가 실제로 data 읽기에 효과적임을 증명한다. 그리고 Directory entry cache 만 사용하는 경우, directory 탐색 비용만 감소하게 되며, buffer cache 만 사용하는 경우보다 더 큰 성능 향상을 보였다. 이는 본 프로젝트의 file system 이 단순 구조이기 때문에 directory 탐색 비용 비중이 더 크기 때문으로 보인다. 두 개의 cache 를 모두 사용하는 경우, data I/O 비용과 directory 탐색 비용 모두 줄어 가장 큰 성능을 보였다.

6. 소프트웨어 유지보수 (Maintenance)

모듈화 설계 & 확장성
기능을 각각의 함수로 분리하였다. File system 초기화(fs_mkfs), mount(fs_mount), file I/O(fs_cat, fs_write_file), directory 관리(fs_ls), cache 관리(lookup_dentry_cache, read_block_with_cache 등)를 모두 독립적인 함수로 구현함으로써 기능 간 결합도를 낮췄다. 단일 책임 원칙 (SRP, Single Responsibility Principle)을 준수하여 코드 수정 시 다른 함수에 대한 영향을 최소화했다.
에러 처리
Perror 등의 함수를 통해 인자가 없거나, 잘못되었거나, 과잉한 경우를 대비하는 시험 & 디버깅을 충분히 실행하였다.
코드 가독성
모든 모듈과 함수에 대한 주석을 작성하여 코드 이해도를 높였다.

7. 결론

본 프로젝트를 통해 file system 이 super block, i-node, data block 으로 구성되어 논리적 file 구조를 물리적 저장 공간에 mapping 하는 과정을 이해할 수 있었다. 구현 과정에서 file 탐색 흐름과 cache 처리 방법을 설계하는 데 어려움이 있었으나, 이를 통해 운영체제 내부 동작에 대한 이해가 깊어졌다. 반면, bitmap 기반 자원 관리, 다중 사용자 환경을 위한 동시성 제어 등 실제 file system 과의 차이점이 부족한 점이자 개선할 점이라고 생각한다.