Project #8: 하드 디스크 드라이버 추가, 파일시스템 구현, C표준 입출력함수 추가 및 서브디렉토리 추가 구현

신정은, 조재호, 조한주
School of Software, Soongsil University

1. 구현 목표

1.1 하드디스크 드라이버 추가

파일 시스템을 구현하는데 기반이 되는 저장 매체를 제어하는 디바이스 드라이버를 작성한다. 드라이버는 SATA가 아닌 PATA 방식의 디스크 드라이버를 구현한다. PATA 방식은 프로세서의 I/O 포트를 통해 직접 컨트롤러에 접근할 수 있으므로, 코드가 아주 간단하다. QEMU 가상머신 또한 PATA 방식을 지원하기 때문에 테스트하는데 유리하기도 하다. 따라서 하드 디스크 컨트롤러를 제어하여 디스크 정보를 추출하는 방법과 섹터를 읽고 쓰는 방법을 알아보고, 하드디스크의 인터럽트를 활성화하여 인터럽트 기반의 하드디스크 드라이버를 실제로 구현해 본다.

2.2 간단한 파일 시스템 구현

컴퓨터의 핵심 기능 중 하나는 데이터 처리이다. 이러한 데이터 처리를 도맡아하는 프로그램은 데이터베이스 시스템이다. 이러한 데이터 베이스 시스템에도 기반에는 파일 시스템이 있다. 파일 시스템은 데이터를 쉽고 빠르게 저장 및 검색할 수 있도록 저장 매체를 관리하는 일종의 규약이다. 규약만 맞다면 OS와 관계없이 데이터를 읽고 쓸 수 있다. 여러 파일 시스템 중 FAT 파일 시스템과 유사한 파일 시스템을 구현한다. 이는 MS DOS부터 사용한 시스템으로 구조가 간단하고 구현하기 쉽다. FAT 시스템은 여러 섹터를 묶은 클러스터 단위로 데이터를 저장한다. 데이터가 크면 여러 클러스터로 나누고 나누어진 클러스터는 체인 형태로 연결해 전체 데이터를 관리한다. 클러스터를 링크로 관리하는 방식은 현재 클러스터와 다음에 연결된 클러스터의 정보만 알면되므로 구현이 쉽고, 파일 시스템을 유지하는데 필요한 메타 데이터의 크기가 작다. 간단하면서 쓸만한 파일 시스템을 만들기 위해 FAT 파일 시스템처럼 클러스터 체인 구조를 사용하는 파일 시스템을 구현한다.

2.3 C표준 입출력 함수 추가

C 표준 입출력 함수의 함수 원형을 분석해 필요한 자료구조를 정의한다. 그리고 C 표준 입출력 함수의 기능을 이해하고 파일 시스템에서 구현한 저수준 함수를 사용하여 같은 기능을 하는 고수준 함수를 구현한다. 파일이나 디렉터리에 관련된 C표준 입출력 함수가 어떻게 구현되었는지 궁금증을 해결하며 fopen(), fread(), opendir(),readdir() 등의 파일과 디렉토리 관련 입출력 함수를 구현해 본다.

2.4 추가 구현

Subdirectory를 구현해본다. 기존의 MINT OS의 파일 시스템은 단일 디렉토리 구조인 루트 디렉토리 안에서만 작업이 가능하도록 하였다. 하지만 현재 존재하는 OS는 subdirectory를 지원하기 때문에 우리가 구현한 MINT OS에서도 subdirectory를 구현하도록 한다.

subdirectory구현을 확인하기 위해 mkdir, rmdir , cd의 명령어를 구현한다. mkdir 명령어를 통해 디렉토리를 생성하면 해당 디렉토리 안에는 .(dot), ..(dot-dot) 디렉토리가 자동으로 생성된다. cd 명령어를 통해 .(dot)으로 이동하면 현재 디렉토리 이동, 즉 아무 변화가 없고 ..(dot-dot)으로 이동하면 부모 디렉토리로 이동된다. 또한 현재 디렉토리가 어디인지 알기 위해 쉘 프롬프트에 현재 디렉토리의 위치도 같이 표시되도록 한다.

2. 구현 내용

2.1 하드디스크 드라이버 추가

2.1.1 하드디스크 구조, 컨트롤러의 구조와 기능

하드 디스크 디바이스 드라이버를 만들기 전, 하드디스크가 어떤 방식으로 동작하고 구성되는지 살펴본다.

2.1.1.1 하드디스크 구조

하드디스크는 디스크를 저장매체로 사용하며, 고밀도의 디스크 여러 장이 겹쳐진 원통 형태로 구성된다. 각 디스크는 양면으로 기록할 수 있고, 디스크마다 섹터 수와 트랙 수는 모두 같다. 디스크 여러 장을 수직으로 쌓아 트랙이 같은 축에 위치하여 원통처럼 되어 트랙을 실린더라고 부른다. 하드디스크는 디스크를 직접 제어해 처리하므로 디스크를 회전시키는 스핀들 모터, 접근할 섹터나 실린더로 액세스 암을 이동시키는 액추에이터, 액세스 암 등이 있다. 엑세스 암의 끝에는 헤드가 존재하며 헤드 아래에 위치하는 영역의 정보를 읽고 쓴다. 이렇게 우리가 하드 디스크의 파일을 읽고 쓰거나 삭제하면 하드 디스크는 물리적으로 움직여 해당 위치로 이동한 뒤 작업을 수행한다. 디스크에 따라 실린더, 헤드, 섹터 수가 다르므로 임의의 섹터에 접근하려면 세가지 값을 참조하여 실제 디스크의 어드레스로 변환해야 한다.

2.1.1.2 CHS 어드레스 방식과 LBA 어드레스 방식

임의의 섹터 어드레스를 물리적인 어드레스로 변환하기 위한 2가지 방식이 있다.

소프트웨어의 입장에서는 연속된 섹터 번호로 접근하는 것이 훨씬 편리하므로 논리적인 섹터 번호로 접근하려면 하드 디스크가 물리적으로 헤더, 실린더, 섹터로 구성되기 때문에 이 구성에

맞춰 디스크 어드레스로 변환하는 작업이 필요하다. 이러한 방식을 CHS 어드레스 방식이라고 한다. 이 방식은 섹터->헤드->실린더의 순서로 증가시키며 논리적인 섹터 번호를 할당한다.

CHS방식과 달리 하드 디스크의 구성과 관계없이 논리적 섹터 번호로 지정하는 방식을 LBA 어드레스 방식이라고 한다. 디스크의 구성에 구애받지 않고 논리적인 어드레스 섹터로 접근할 수 있는 LBA방식을 근래에는 많이 사용하므로 우리의 MINT64 OS도 LBA어드레스 방식으로 디바이스 드라이버를 작성한다.

2.1.1.3 하드 디스크 컨트롤러, I/O포트, 레지스터

하드 디스크 컨트롤러의 구조를 알아보고, 하드 디스크 컨트롤러에 할당된 레지스터를 살펴보며 하드 디스크를 제어하는 방법을 알아본다.

하드 디스크 컨트롤러는 PC내부 버스와 포트 I/O방식으로 연결되어 있으며, 사용할 수 있는 PATA포트는 최대 두 개이다. 첫 번째 PATA포트가 0x1F0~0X1F7, 0X3F6을 사용하고, 두번째 PATA포트가 0x170~0x177과 0x376을 사용한다. 각 PATA포트는 마스터와 슬레이브로 두 개의 하드 디스크를 연결할 수 있으니 PC에 연결할 수 있는 하드 디스크는 최대 4개이다.

커맨드 레지스터는 하드 디스크로 전송할 커맨드를 저장하는 레지스터이다. 하드 디스크는 커맨드를 전송하는 순간 다른 레지스터가 모두 설정되었다고 가정하고 작업을 시작한다. 따라서 커맨드 레지스터는 가장 마지막에 설정한다. 읽기 커맨드는 하드 디스크로부터 섹터를 읽는 기능을 한다. 만약 읽는 도중 문제가 발생하면, 하드디스크가 자체적으로 여러 번 시도하며, 모두 실패한 경우 상태 레지스터에 에러를 설정한다. 쓰기 커맨드는 하드 디스크에 데이터를 쓴다는 점을 제외하면 읽기 커맨드와 같은 방식으로 동작한다. 드라이브 인식 커맨드는 하드 디스크 드라이브에 관련된 정보를 넘겨준다. 데이터 레지스터를 통해 하드디스크 정보를 읽을 수 있다. 하드 디스크는 작업이 완료되면 디지털 출력 레지스터의 인터럽트 플래그를 참조하여 인터럽트를 발생시킨다. 데이터 레지스터는 커맨드 수행이 완료되었을 때 하드 디스크로 데이터를 전송하거나 하드 디스크에서 데이터를 읽을 때 사용한다. 데이터 레지스터는 다른 레지스터와 달리 크기가 2바이트이므로 WORD단위 레지스터로 접근해야 한다.

섹터 수 레지스터와 섹터 번호 레지스터, 실린더 레지스터와 드라이브/헤드 레지스터의 크기는 모두 1바이트이며 읽기,쓰기 커맨드 수행 시 하나의 세트처럼 사용된다. 하드디스크는 이 세트 레지스터가 지정하는 섹터를 시작으로 섹터 수 레지스터에 저장된 값만큼 데이터를 읽거나 쓰는 방식으로 동작한다. 섹터 수 레지스터는 처리할 섹터 수를 나타내는 레지스터이다. 섹터 번호 레지스터, 실린더 레지스터, 드라이브/헤드 레지스터의 헤드 필드는 작업을 시작할 섹터 어드레스를 저장하는 레지스터이다. 섹터 수 레지스터를 제외한 세가지 레지스터는 CHS모드와 LBA모드일 때 역할이 다르다. LBA모드에서는 레지스터가 LBA어드레스로 통합되어 섹터 번호 레지스터는 0~7비트, 실린더 LSB레지스터는 8~15비트, 실린더 MSB레지스턴S 16~32비트, 드라이브/헤드 레지스터의 헤드 필드는 24~27비트를 저장한다. 드라이브/ 헤드 레지스터에는 LBA 모드 필드와 드라이브 번호 필드가 더 있다. LBA 모드 필드는 어드레스 모드를 설정하는 필드로, 1로 설정하여 LBA 어드레스 모드로 동작하게 한다. 드라이브 번호 필드는 PATA포트에 연결된 마스터나 슬레이브 하드 디스크 중 어느 것을 선택할지 나타내는 필드이다. 0으로 설정하면 마스터 하드디스크로 커맨드가 전송되고 1로 설정하면 슬레이브 하드 디스크로 커맨드가 전송된다. 지금까지 살펴본 4개의 레지스터는 하드 디스크가 처리할 정보를 나타내므로 반드시 커맨드를 전송하기 전 먼저 설정해야 한다.

상태 레지스터는 하드 디스크의 작업 상태를 나타내는 1바이트 크기의 레지스터로 우리가 전송한 커맨드의 실행 결과와 에러 유무를 나타낸다. BSY비트를 통해 데이터를 설정할 시점을 판단할 수 있다. 그리고 커맨드는 BSY필드가 0이고 DRDY필드가 1일때만 송신할 수 있고, DRQ필드가 1인지 검사하여 데이터를 송수신 할 수 있는지 확인해야 한다. 이 외에도 IRQ 14,15를 통해 인터럽트 작업이 끝났음을 알 수 있고, ERR필드로 에러 여부를 알 수 있다.

하드 디스크 컨트롤러도 인터럽트를 발생시킬 수 있는데, 인터럽트 발생 여부는 디지털 출력 레지스터가 담당한다. 이 외에도 하드디스크 리셋 기능도 담당한다. SRST필드가 1이면 포트에 연결된 모든 디바이스를 리셋한다. 0으로 설정 되어야만 정상 동작하고, 1이면 인터럽트가 발생하지 않는다.

2.1.2 하드 디스크 디바이스 드라이버 설계와 구현

위의 레지스터를 사용하여 하드 디스크를 제어하기 위해 우선 자료구조를 설계한 후 세부 함수를 구현한다.

2.1.2.1 디바이스 드라이버 설계

MINT64 OS는 PATA방식의 디바이스 드라이버를 구현하므로 PATA방식의 하드 디스크가 존재하는지 저장하는 필드를 두어 수행 여부를 확인해본다. 그리고 실제 하드디스크가 아닌 가상 머신에서 실행될 때만 하드 디스크를 쓸 수 있도록 제한하는 필드도 추가한다, 그리고 인터럽트 발생 여부를 저장할 플래그와 태스크 동기화 문제 해결을 위한 동기화 객체도 추가한다.

하드디스크를 관리하는 구조체 HDDMANAGER에서 뮤텍스를 사용한 이유는 하드 디스크는 섹터 단위로 처리해서 커맨드를 전송하고 데이터를 송/수신하는데 시간이 많이 걸리기 때문이다. 또한 뮤텍스를 사용하면 동기화 함수와 같은 기능을 하며, 임계영역에서 인터럽트를 처리할 수 있는 장점이 있다. StHDDInformation 필드는 하드 디스크 정보를 나타내는 구조체로 하드디스크 인식 커맨드를 사용해서 읽을 수 있다. 우리는 LBA 어드레스 모드를 사용하므로 하드 디스크의 총 섹터 수와 모델 번호만 알면 된다.

2.1.2.2 디바이스 드라이버 초기화

하드 디스크를 초기화하는 함수는 변수와 뮤텍스를 초기화하고, 하드 디스크의 정보를 추출하여 하드 디스크 자료구조에 삽입한다. 하드 디스크 인식 커맨드를 통해 하드 디스크 정보가 나왔다면, 하드디스크 모델 번호를 이용해 QEMU에서 실행되는지 알 수 있다. PATA방식의 하드디스크는 최대 4개를 연결할 수 있지만 첫 번째 PATA 포트에 마스터로 하드 디스크를 연결하므로 해당 위치만 검색해도 된다. 이렇게 초기화 함수를 수행하고 나면 하드 디스크가 존재하는 지와 쓰기가 가능한지 결정할 수 있게 된다.

2.1.2.3 하드 디스크 정보 추출

하드 디스크의 정보를 추출하는 함수는 PATA 포트 위치, 마스터, 슬레이브 정보를 파라미터로 넘겨받아 하드 디스크 정보를 반환하는 기능을 한다. 크게 드라이브 정보를 설정하는 부분, 커맨드를 전송한 후 인터럽트를 대기하는 부분, 데이터 포드에서 하드 디스크 정보를 읽는 부분으로 나눌 수 있다.

드라이브 정보를 설정하는 부분에서 중요한 것은 하드 디스크의 작업이 끝나고 난 후 레지스터에 드라이브 정보를 설정해야 한다는 것이다. 따라서 상태 레지스터의 비트 7이 1인지 검사하고, Busy하지 않다면 드라이브 정보를 추출한다. 드라이브를 선택한 후엔 커맨드를 전송하고 인터럽트가 발생할 때까지 대기한다. 이때는 하드 디스크가 Busy상태가 아니며 동시에 Ready상태여야 한다. Ready상태를 기다린 뒤 커맨드를 전송하면 커맨드 수행이 완료된 뒤에 인터럽트가 발생한다. 인터럽트가 발생하면 하드디스크 자료구조의 bPrimaryInterruptOccur필드와 bSecondaryInterruptOccur필드가 TRUE로 설정된다.

하드 디스크에서 데이터를 읽을 때에는 IN 어셈블리어 명령으로 데이터 레지스터를 반복해서 읽는다. 데이터 레지스터는 다른 레지스터와 달리 2바이트므로 AX레지스터를 사용하여 2바이트 단위로 읽고 쓰는 I/O포트 입출력 함수를 작성한다.

2.1.2.4 섹터 읽기

섹터 읽기는 하드 디스크에서 1섹터부터 최대 256 섹터까지 데이터를 읽는다. 섹터 읽기 함수는 섹터 수 레지스터, 섹터 번호 레지스터, 실린더 레지스터를 설정하는 부분을 제외하면 하드 디스크 정보를 읽는 함수와 거의 같다. LBA섹터 어드레스를 나누어 각종 레지스터를 설정한 후 커맨드를 전송한다. 하드 디스크는 데이터가 준비 됐을 때 인터럽트를 발생시키므로 여러 섹터를 읽는 경우를 대비해 데이터 수신 코드에서 인터럽트를 대기한다. 또한 섹터에서 에러가 날 수도 있기 때문에 매번 상태 레지스터를 읽어 에러 발생 여부를 확인한다.

2.1.2.5 섹터 쓰기

섹터 쓰기 함수는 섹터 읽기 함수와 반대로 하드 디스크에 섹터를 쓰고 난 뒤에 인터럽트가 발생한다. 따라서 상태 레지스터의 DATAREQUEST비트가 설정되기를 기다린 후 데이터를 송신 가능한 상태로 만든다. 인터럽트가 발생하는 시점의 차이 때문에 인터럽트를 대기하는 코드와 상태를 검사하는 코드가 한 섹터를 전송한 다음으로 위치하게 된다.

2.1.2.6 인터럽트 처리

하드 디스크 인터럽트를 처리하기 위해 인터럽트 핸들러를 작성한 후 이를 ISR함수에서 호출하도록 한다. 하드 디스크 인터럽트 핸들러에서는 인터럽트 발생 여부를 화면에 표시하고, PIC 컨트롤러에 EOI를 전송한다. 그리고 인터럽트가 발생할 때 마다 하드 디스크 자료구조에 인터럽트 발생 필드를 업데이트 한다. ISR함수에서는 첫 번째 PATA 포트에서 인터럽트가 발생했을 때 작성한 핸들러가 호출되도록 한다.

2.1.3 빌드 및 구동 확인

HardDisk.c/h 파일에는 하드 디스크 디바이스 드라이버를 초기화하는 함수, 하드 디스크의 정보를 추출하는 함수, 섹터를 읽고 써서 데이터를 송수신하는 함수와 인터럽트 발생 여부를 설정하고 발생할 때까지 대기하는 함수를 작성한다. AssemblyUtility.asm/h 파일에는 I/O포트에서 2바이트씩 데이터를 읽고 쓰는 함수를 추가한다. 그리고 InterruptHandler.c/h함수를 수정하여 하드 디스크에서 발생하는 인터럽트 핸들러 함수를 작성한다. 그리고 ISR.asm파일을 수정해 첫 번째 PATA 포트의 인터럽트와 두 번째 PATA포트의 인터럽트를 처리하는 함수를 수정한다. Main.c에서는 하드 디스크 초기화가 완료되었는지 출력해주는 부분을 추가한다. 콘솔 셸 파일을 수정하여 작성된 드라이버를 테스트하는 hddinfo,writesector, readsector 커맨드를 추가한다. Hddinfo는 하드 디스크의 정보를 출력하고, writesector는 LBA어드레스와 섹터 수를 파라미터로 받아 하드 디스크에 데이터를 쓴다. 쓰기가 정상적으로 수행되었는지 확인하기 위해 readsector 커맨드를 사용한다. 마지막으로 MAKEFILE을 수정하여 QEMU용 하드 디스크 이미지를 생성하고 QEMU에 하드 디스크를 추가한다.

MINT64 OS를 실행하면 초기화가 완료되고, 별다른 이상이 없다면 hddinfo입력 시 모델 번호에 QEMU HARDDISK가 표시되고 전체 섹터 수가 40960섹터로 표시된다. readsector 0 2를 입력해 LBA 0 어드레스를 시작으로 2 섹터를 읽어보면 모두 0으로 나온다. 섹터 읽고 쓰기를 반복하여 하드 디스크 드라이버가 정상적으로 동작하는지 확인할 수 있다. 이제 하드 디스크 드라이버를 기반으로 파일 시스템을 구현해 보자!

2.2 파일 시스템 구현

2.2.1 파일 시스템 설계

파일 시스템의 구조와 각 영역을 구성하는 자료구조를 작성한다.

2.2.1.1 파일 시스템의 특징

이번에 구현할 파일 시스템은 메타 정보의 크기를 줄이고 하드 디스크에 더 빠르게 접근하

려고 4KB를 묶어서 하나의 클러스터로 사용한다. 그리고 복잡함을 줄이기 위해 수평적인 구조로 설계한다. 루트 디렉터리에 최대 128개의 파일을 생성할 수 있고, 약 4GB까지 단일 파일을 지원하도록 만든다.

2.2.1.2 파일 시스템 구조

실제 데이터를 저장하는 데이터 영역과 이를 관리하는데 필요한 메타 데이터 영역으로 구분된다. 메타 데이터 영역은 다시 MBR 영역, 예약된 영역, 클러스터 링크 테이블 영역으로 나뉜다. 메타 데이터 영역은 데이터 영역의 비어 있는 공간과 파일을 구성하는 클러스터의 연결 관계를 관리한다. 메타 정보 중 클러스터 링크 테이블 영역이 핵심이다. 이는 해당 클러스터의 할당 여부를 저장하고, 할당되었다면 다음에 연결된 클러스터의 인덱스를 저장한다. 또 데이터를 추가, 삭제, 읽을 때 사용된다.

2.2.1.3 MBR 영역

MBR 영역은 하드 디스크의 첫번째 섹터이며, 부트 코드, 파티션 정보와 파일 시스템 정보가 들어 있다. 파티션은 하드 디스크를 논리적으로 분할한 영역이다. 주로 하드 디스크 하나에 여러 OS를 설치하거나 OS와 데이터 영역을 구분할 목적으로 사용한다. 파티션은 MBR 영역의 뒤쪽에 있는 파티션 테이블 영역에 정보를 기록하여 생성한다. MBR 시작 어드레스부터 446 바이트 떨어진 위치에 존재하며, 총 4개의 파티션 정보, 즉, 부팅 가능 여부, 시작과 끝, 파티션의 종류, 총 섹터를 담고 있고 크기는 16 바이트이다. 파티션을 생성할 때 서로 겹치는 부분이 없도록 해야한다.

파티션 테이블은 선택 사항으로 이번에 우리도 구조는 만들지만 테이블을 모두 0으로 지워준다. MBR의 시작 위치부터 파티션 테이블까지 44 바이트는 부트 로더 코드와 파일 시스템 데이터가 위치한다. MINT 64 OS 는 그 중에서 파티션 테이블 이전의 영역을 일부 사용한다. 파일 시스템 시그니처, 예약된 영역의 크기, 클러스터 링크 테이블 영역의 크기, 클러스터의 개수 등이다.

2.2.1.4 예약된 영역

파일 시스템의 예약된 영역은 선택 사항으로 파일 시스템 백업 공간이나 외부로 공개할 수 없는 데이터를 저장하는 영역으로 사용한다. MINT 64 OS는 OS 이미지가 디스크의 첫 번째 섹터부터 연속적으로 저장되어 예약된 영역이 유용하다. 파일 시스템을 생성할 때 OS 이미지 크기만큼 예약된 영역으로 설정해 파일 시스템 데이터와 영역이 겹치지 않도록 한다. 파일 시스템 확장을 위해 추후 사용한다.

2.2.1.5 클러스터 링크 테이블 영역과 데이터 영역

클러스터 링크 테이블을 구성하는 링크 정보는 클러스터 수 만큼 존재하며, 링크 정보의 인덱스는 데이터 영역 내에 클러스터의 인덱스와 같다. 링크 정보의 크기는 4 바이트로 필요한 클러스터 링크 테이블의 크기는 4 바이트와 클러스터의 개수의 곱이다. 하드 디스크는 섹터 단위로 데이터를 처리하여, 실제 하드 디스크에서 차지하는 영역은 계산된 크기를 섹터 단위로 올림한 크기이다.

데이터 영역은 메타 정보가 차지하는 영역을 제외한 나머지 영역으로 전체 크기는 클러스터 단위로 정렬되어 있다. 파일도 클러스터 단위로 처리한다.

클러스터 링크 테이블의 링크 정보를 보고 파일의 클러스터를 찾는다. 클러스터의 인덱스와 링크 정보의 인덱스는 1:1로 대응하며, 각 링크 정보는 클러스터의 사용 여부와 연결된 클러스터의 오프셋을 가진다.

파일의 시작 클러스터 주소는 디렉터리가 갖고 있는다. MINT 64 OS의 파일 시스템은 가장 첫 번째 클러스터 0을 최상위 디렉터리인 루트 디렉터리로 사용한다.

2.2.1.6 루트 디렉터리 파일

디렉터리는 파일이지만, 파일을 관리하는 데 사용하기 때문에 다르게 처리한다. 현재까지는 루트 디렉터리만 사용하여 모든 파일을 수평적으로 관리한다.

루트 디렉터리로 첫 번째 클러스터 0을 예약해 사용하고 루트 디렉터리는 엔트리로 이루어져 있다. MINT 64 OS의 클러스터 크기는 4KB로 디렉터리 엔트리가 최대 128개 생성된다. 엔트리에는 파일 이름과 파일 크기, 시작 클러스터 정보가 들어있다.

2.2.1.7 파일 추가와 삭제 알고리즘

루트 디렉터리에 파일을 추가하려면 디렉터리 엔트리 1개와 클러스터 1개가 필요하다. 루트 디렉터리를 우선 검색하여 빈 엔트리를 확인하고 클러스터 링크 테이블을 검색해 빈 클러스터를 찾는다. 둘을 찾으면 해당 클러스터의 링크 테이블에 0xFFFFFFFF를 설정해 클러스터가 할당되고 마지막 클러스터임을 표시한다. 그리고 디렉터리 엔트리에 파일 이름, 크기, 할당받은 클러스터 인덱스를 저장한다.

파일 삭제는 루트 디렉터리에서 같은 이름의 디렉터리 엔트리를 찾아 클러스터 링크 테이블을 0x00으로 설정한다. 크리고 디렉터리 엔트리를 0으로 초기화한다.

2.2.1.8 파일 시스템 자료구조 설계

파일 시스템의 자료구조에는 파일 시스템 인식 여부, 각 영역의 섹터 수와 시작 LBA 어드레스, 클러스터의 총 개수, 클러스터를 할당한 클러스터 링크 테이블의 섹터 오프셋와 파일 시스템 동기화 객체가 있다. 파일 시스템을 초기화시 bMount 필드를 확인해 TRUE일 때만 파일 시스템을 정상적으로 사용한다.

2.2.2 저수준 함수 구현

파일 시스템 초기화 함수, 파일 시스템 생성 함수, 클러스터 링크 테이블 제어 함수, 루트 디렉터리 제어 함수를 구현한다.

2.2.2.1 파일 시스템 초기화 함수와 함수 포인터 설정

파일 시스템 자료구조를 초기화하고, 모듈이 사용하는 함수의 포인터를 설정하는 함수를 작성한다. 하드 디스크에 종속되지 않는 코드로 모듈 초기화 함수에서 함수 포인터를 변경하도록 하면 상황에 따라 저장 매체를 동적으로 바꾸도록 한다.

하드 디스크 초기화를 수행하여 하드 디스크가 정상적으로 인식되어야 함수 포인터를 등록하도록 한다. kMount() 함수는 하드 디스크의 파일 시스템이 MINT 64 OS의 파일 시스템인지 확인한다.

2.2.2.2 파일 시스템 인식 함수

파일 시스템이 MINT 64 OS의 파일 시스템인지 확인하려면 MBR 영역의 뒷부분에 있는 파일 시스템 시그니처를 검사한다. 파일 시스템 각 영역의 시작 정보와 섹터 수는 시그니처 뒷부분에 있는 값을 이용하여 계산한다.

2.2.2.3 파일 시스템 생성 함수

하드 디스크의 크기에 맞춰 메타 영역과 데이터 영역을 생성한다. 이는 포맷이라고도 한다. 이를 kFormat()으로 작성한다. 하드 디스크의 총 섹터 수를 이용하여 메타 영역의 크기를 계산하는 부분과 계산된 정보를 이용하여 루트 디렉터리까지 초기화하는 부분으로 나눈다.

2.2.3 클러스터 링크 테이블 관련 함수 작성

빈 클러스터를 찾는 함수와 클러스터 링크 테이블에 링크 정보를 설정하는 함수, 클러스터 링크 테이블에서 링크 정보를 읽는 함수를 구현한다.

2.2.3.1 클러스터 링크 테이블 읽기 함수와 쓰기 함수

클러스터 링크 테이블 내의 섹터 오프셋으로 접근하여 읽고 쓴다. 함수는 파일 시스템 자료구조에 저장된 클러스터 링크 테이블의 시작 어드레스를 더하는 역할을 한다.

2.2.3.2 빈 클러스터 검색 함수

클러스터 링크 테이블 영역을 돌면서 빈 클러스터로 표시된 링크 정보를 찾는다. 읽은 섹터 내애서 값이 0x00으로 저장된 부분을 찾으면 된다.

2.2.3.3 링크 정보 반환 함수와 설정 함수

클러스터 링크 테이블에서 해당 클러스터의 링크 정보를 반환하거나 설정한다. 클러스터 인덱스로 링크 정보를 반환하려면 링크 정보가 포함된 섹터의 오프셋과 해당 섹터 내의 오프셋이 필요하다.

2.2.4 루트 디렉터리와 파일 관련 함수

클러스터 링크 테이블 관련 함수와 구조는 같지만 대상이 데이터 영역의 클러스터이며 링크 정보가 아닌 디렉터리 엔트리이다.

2.2.4.1 클러스터 읽기 함수와 쓰기 함수

데이터 영역의 클러스터 오프셋으로 접근하여 읽고 쓴다. 데이터 영역의 시작 어드레스를 더하는 역할만 한다.

2.2.4.2 빈 디렉터리 엔트리 검색 함수와 디렉터리 엔트리 반환/설정 함수

앞서 구현한 클러스터 링크 테이블 함수와 형태가 비슷하다. 대상이 데이터 영역의 클러스터와 링크 정보가 아닌 디렉터리 엔트리라는 것이 다르다.

2.2.5 파일 추가 함수와 삭제 함수

후에 구현한 고수준 함수를 검증하기 위해 임시 함수를 작성한다.

파일 생성은 클러스터 할당, 디렉터리 엔트리 할당, 디렉터리 엔트리 등록 순서로 작성한다.

파일 삭제는 디렉터리 엔트리 검색, 클러스터 반환, 디렉터리 엔트리 삭제의 순서로 작성한다. 파일 추가와 달리 예외가 발생하면 다시 상태를 복원하지는 않고 에러 메시지만 출력한다.

2.2.6 빌드 및 구동 확인

FileSystem.c/h 에는 파일 시스템 초기화 함수부터 클러스터 링크 테이블, 루트 디렉터리 관련 함수, 파일 시스템 생성 함수를 포함한다. 그리고 Main.c에 파일 시스템을 초기화하는 코드를 추가한다. 마지막으로 콘솔 셸 파일에 파일 시스템을 연결하는 mounthdd, 파일 시스템을 생성하는 formathdd, 파일 시스템 정보를 표시하는 filesysteminfo, 파일을 생성하고 삭제하는 createfile, deletefile, 루트 디렉터리의 파일을 확인하는 dir 커맨드를 추가한다. 그리고 hddinfo 커맨드를 수정하여 파일 시스템 모듈의 함수를 호출하도록 한다.

코드를 작성하고 빌드하면 처음에 파일 시스템을 인식하지 못하여 File System Initialize가 fail이 된다. Formathdd와 mounthdd 커맨드를 입력하면 파일 시스템이 생성되고 연결된다. 이제 createfile과 deletefile 명령어를 통해 파일의 추가/삭제를 확인할 수 있다. 그리고 filesysteminfo를 통해 각 영역의 정보와 데이터 영역의 총 클러스터 개수를 확인할 수 있다. 이제 다음에는 C 언어의 표준 입출력 함수와 같은 고수준의 함수를 작성한다.

2.3 C 표준 입출력 함수 추가

2.3.1 C표준 입출력 함수 설계

C표준 입출력 함수를 살펴보기 전, MINT64 OS에서 구현할 함수가 어떤 타입으로 정의되어 있는지 함수 원형을 먼저 확인한다. 그리고 각 함수가 사용하는 자료구조를 살펴보고 이를 바탕으로 MINT파일 시스템에 해당 자료구조를 추가해 본다.

2.3.1.1 구현할 함수 목록과 함수 원형

C표준 라이브러리를 따르기로 했으므로 먼저 해당 함수의 원형을 살펴본다. FILE과 DIR 타입을 사용해 파일과 디렉터리를 처리하고 있다. 두 타입은 파일과 디렉터리의 상태를 저장하는 자료구

조이다. 커널 내부에서 사용하는 자료구조와 유저 레벨에서 사용하는 자료구조를 통일하면 표준 입출력 함수를 보다 쉽게 구현할 수 있을 것이다. 따라서 두 자료구조를 어떻게 활용할 지 살펴본다.

2.3.1.2 FILE 자료구조와 DIR 자료구조 설계

우선 FILE자료구조를 먼저 정의한다. 표준 입출력 함수처럼 파일을 클러스터 단위가 아닌 바이트 단위로 처리하려면 현재 작업이 수행되는 위치에 따라 클러스터를 이동하는 부분이 필요하다. 이러한 작업을 처리하기 위해 파일 포인터의 현재 위치와 파일 시작 클러스터 인덱스, 파일 포인터가 위치한 클러스터의 인덱스가 필요하다. 그리고 파일의 끝을 판단하기 위해 실제 파일의 크기도 필요하다. 그리고 디렉터리 엔트리의 오프셋을 저장해 파일의 크기가 커질 때 마다 해당 오프셋만 갱신하도록 한다.

디렉터리는 파일보다 간단하다. 디렉터리 관련 함수는 순차적으로 엔트리를 읽는 기능만 제공하므로, 루트 디렉터리에서 현재 어느 오프셋의 엔트리를 읽고 있는지 저장한다. 또한 디렉터리 관련 함수는 루트 디렉터리에 접근하는 횟수가 많으므로 루트 디렉터리의 내용을 버퍼에 저장한다.

두 자료구조를 만들었으니, 위에서 말한 것처럼 관리를 위해 두 자료구조를 Union을 사용해 하나의 자료구조로 합친다.

2.3.2 핸들과 파일 관련 고수준 함수 구현

2.3.2.1 파일 시스템 초기화 함수 수정

초기화 함수에서 파일 시스템의 핸들 자료구조를 처리해 준다. 초기화 하기 전 핸들을 위한 공간을 확보한다. 동적 메모리를 사용하는 것이 확장에 유리하므로 핸들 풀은 동적 메모리를 할당 받아 사용한다. 핸들의 개수는 최대 태스크 개수의 3배로 지정한다.

2.3.2.2 핸들 할당 함수와 해제 함수

핸들의 타입은 파일 타입과 디렉터리 타입, 할당되지 않은 상태를 나타내는 빈 타입 세가지로 구분된다. 최초의 풀은 빈 타입이고, 핸들을 할당하는 함수는 풀을 검색하여 빈 타입으로 설정된 핸들을 찾아 반환하고 할당된 것으로 표시한다. 핸들을 해제하는 함수는 반대로 주어진 핸들에 핸들 타입을 빈 타입으로 설정한다.

2.3.2.3 파일 열기 함수 - fopen()

파일을 열 때 수행할 작업은 mode 파라미터에 의해 결정된다. mode에는 r, w, a 등이 있다. 파일 열기 함수는 mode에 따라 파일 생성, 비움, 파일 포인터 이동 세가지 기능을 수행해야 한다. 각 부분은 디렉터리 엔트리의 존재 여부와 mode에 따라 선택적으로 수행된다. 파일 열기 함수에서 가장 먼저 하는 작업은 루트 디렉터리에 디렉터리 엔트리가 존재하는지 확인하여 파일을 생성하거나 비우는 일이다. 만약 파일이 없는데 mode가 w와 a 옵션이라면 파일을 생성하고 w 옵션일 경우 두 번째 클러스터부터 마지막 클러스터까지 모두 해제하여 파일을 모두 비운다. 그리고 파일의 길이를 0으로 설정하고 디렉터리 엔트리를 업데이트 하여 파일 길이를 0으로 설정한다. mode에 따라 파일을 생성하거나 비우는 작업을 한 후, 파일 핸들을 할당받아 파일의 정보를 설정한다. 파일 정보는 디렉터리 엔트리에 대부분 들어있으므로 값을 그대로 파일 핸들에 저장하면 된다. mode가 a라면 마지막 클러스터를 찾아 파일 포인터를 끝으로 이동시키는 함수를 호출한다.

2.3.2.4 파일 읽기 함수 - fread()

파일 포인터를 이동시키지 않는 한 파일의 시작부터 끝 방향으로 파일 포인터가 이동한다. 따라서 파일 포인터를 따라 정확한 클러스터를 찾아 나가는 것이 파일 읽기, 쓰기 함수의 핵심이다. MINT 파일 시스템은 클러스터 단위로 데이터를 저장하므로, 요청된 데이터가 여러 클러스터에 걸쳐 있을 수 있다. 이 때 클러스터를 순차적으로 이동하며 복사할 크기를 계산하여 나누어 처리해야 한다.

2.3.2.5 파일 쓰기 함수 - fwrite()

fread()함수와 마찬가지로 레코드의 크기와 개수를 곱한 크기만큼 buffer에서 파일로 쓴다. 파일을 확장하는 부분과 파일 크기를 업데이트 하는 부분을 제외하면 파일 읽기 함수와 비슷하므로 다른 부분을 살펴본다. 클러스터에 쓰다 보면 파일의 끝부분에 도달해 클러스터를 더 할당해야 하는 경우가 생긴다. 이때는 더 쓸 공간이 없으므로 새로운 클러스터를 할당 받아 파일을 확장해야 한다. 빈 클러스터를 할당 받은 뒤, 파일의 마지막 클러스터로 설정하고 현재 클러스터의 클러스터 링크의 값을 할당 받은 클러스터의 인덱스로 설정한다. 그리고 디렉터리 엔트리의 정보를 변경하여 파일의 크기를 업데이트 한다.

2.3.2.6 파일 포인터 이동 함수 - fseek()

파일 포인터의 위치는 기준점이나 원점을 나타내는 origin값과 offset에 따라 달라진다. origin값에는 SEEK_SET, SEEK_CUR, SEEK_END 세가지가 있다. 가장 먼저 파일의 시작을 기준으로 하는 실제 파일 오프셋을 계산하고 클러스터 링크를 따라가며 파일 포인터가 위치하는 클러스터까지 이동한다. 만약 파일 포인터의 값이 파일 크기보다 크다면 마지막 클러스터까지 이동하여 파일 끝으로 설정한 뒤, 실제 파일 오프셋까지 남는 부분을 0으로 채워준다. 다 이동했다면 파일 포인터의 위치를 갱신한다.

2.3.2.7 파일 닫기 함수 - fclose()

파일 닫는 함수는 파라미터가 핸들타입인지 검사하고, 파일 핸들을 반환하면 된다.

2.3.2.8 파일 제거 함수 - remove()

파일 제거 함수는 디렉터리 엔트리를 직접 제어하여 파일을 삭제한다. 파일 삭제를 위해선 두가지를 고려해야 하는데, 첫째는 삭제할 파일을 사용하는 태스크가 없어야 한다는 것이다. 이를 위해 다른 태스크가 같은 파일이 열려있는지 핸들풀을 검색하여 판단한다. 파일 핸들의 필드 중 시작

클러스터가 일치하는 핸들을 검색하면 된다. 두번째는 파일을 삭제할 때 시작 클러스터부터 마지막 클러스터까지 링크를 따라가며 모두 삭제해야 한다는 것이다. 그리고 루트 디렉터리에서 디렉터리 엔트리를 삭제하여 파일을 지우는 과정은 루트 디렉터리에서 파일 이름과 일치하는 디렉터리 엔트리를 검색한 뒤에 모두 0으로 초기화하여 덮어쓰면 된다.

2.3.3 디렉터리 관련 고수준 함수 구현

2.3.3.1 디렉터리 열기 함수 – opendir()

원래 경로에 위치하는 디렉터리를 열어 핸들을 반환한다. 원래는 이름에 해당하는 디렉터리를 찾지만, MINT64 OS는 루트 디렉터리밖에 지원하지 않으므로 루트 디렉터리의 핸들을 생성하여 반환한다. 파일을 여는 함수와 다른 점은 디렉터리는 클러스터 한 개로 제한되어 있고 디렉터리 관련 함수는 루트 디렉터리의 엔트리를 반복해서 검사하므로 버퍼를 이용하여 루트 디렉터리가 있는 클러스터를 저장한다.

2.3.3.2 디렉터리 읽기 함수와 디렉터리 포인터 되감기 함수 – readdir(), rewinddir()

디렉터리를 읽는 함수는 루트 디렉터리 버퍼에서 디렉터리 엔트리의 오프셋이 가리키는 엔트리를 반환한다. 디렉터리를 되감는 함수는 디렉터리 엔트리의 오프셋을 루트 디렉터리의 처음으로 이동시킨다.

디렉터리를 읽는 함수는 루트 디렉터리에 존재하는 수많은 엔트리 중 시작 클러스터를 확인하여 실제로 유효한 엔트리를 찾는다. 이렇게 유효한 엔트리를 찾은 후엔 해당 엔트리를 반환하여 디렉터리 엔트리 오프셋을 다음으로 옮겨준다.

디렉터리 포인터를 되감는 함수는 디렉터리 핸들에 있는 디렉터리 엔트리 오프셋을 0으로 설정해주는 것이 전부이다.

2.3.3.3 디렉터리 닫기 함수 – closedir()

사용이 끝난 디렉터리 핸들을 반환한다. 그리고 할당 받은 버퍼를 반환한다.

2.3.3.4 함수 이름과 매크로 타입을 C표준 라이브러리 스타일로 변환

지금까지 함수 이름과 자료구조의 이름이 MINT64 OS기준으로 작성되었지만 파일 시스템을 사용하는 프로그래머 입장에서는 표준 입출력 함수 스타일이 편하므로 재정의를 통해 변환한다.

2.3.4 빌드 및 구동 확인

FileSystem.c/h 파일을 수정하여 핸들을 위한 공간을 할당하고 핸들 풀을 초기화한다. 그리고 지금까지 구현한 고수준 함수의 코드를 추가한다. 그리고 헤더 파일에는 함수이름과 매크로 등을 표준 입출력 방식으로 재정의하는 코드도 추가한다. Utility.h파일에는 두 수중 큰 값 또는 작은 값을 반환하는 매크로를 추가하고, 콘솔 셸 파일에 writefile, readfile, testfileio 커맨드를 추가한다. writefile 커맨드는 키보드로 값을 입력받아 1바이트씩 파일에 쓴다. 엔터키가 3번 연속으로 입력되면 파일 입력을 종료하고 파일을 생성한다. readfile 커맨드는 파일이 정상적으로 생성되었는지 확인할 목적으로 사용한다. testfileio 커맨드는 고수준 함수를 사용해 파일 포인터를 이동하면서 썼을 때 원하는 결과가 나오는지 확인하는 것이 목적이다. 파일 생성, 순차 영역 쓰기와 읽기, 임의 영역 읽기와 쓰기, 파일 삭제와 닫기 테스트를 수행한다. 그리고 앞 절에서 작성한 createfile, deletefile, dir 커맨드가 고수준 파일 입출력 함수를 사용하도록 수정한다.

빌드를 하고 커맨드를 테스트한 결과 오류 없이 파일 생성, 쓰기, 읽기, 삭제 등 고수준 함수가 잘 구동 되는 것을 확인할 수 있다. 이렇게 표준 입출력 형식의 함수를 구현하여 편리하게 파일을 읽고 쓸 수 있게 되었다. 그러나 MINT64 OS는 현재 루트 디렉터리만 진행하고 있다. 실제 OS처럼 파일 시스템을 활용하기 위해 다음 절에서는 하위 디렉터리를 추가 구현하고 이를 기반으로 mkdir, rmdir, cd 명령어를 구현해 본다.

2.4 추가구현

2.4.1 쉘에 현재 디렉토리의 위치 구현

Consoleshell 파일에 path라는 전역변수를 만들어 현재 디렉토리 위치를 나타내도록 한다. Cd 명령어를 통해 디렉토리를 이동할 때마다 path 전역변수를 수정해줘서 지속적으로 현재 디렉토리 위치를 나타낸다.

2.4.2 DIRECTORYENTRY 구조체 변경

기존 DIRECTORYENTRY 구조체는 부모 디렉토리의 정보를 가지고 있지 않다. 하지만 subdirectory를 구현하기 위해서는 디렉토리마다 부모 디렉토리의 정보를 가져야 한다. 그래서 구조체에 부모 디렉토리의 경로를 나타내는 ParentDirectoryPath 변수와 부모 디텍토리의 클러스터 인덱스를 나타내는 ParentDirecytoryClusterIndex 변수를 추가한다. 또한 파일과 디렉토리를 구분하기 위해 flag라는 변수를 사용한다.

```
// 디렉터리 엔트리 자료구조
typedef struct kDirectoryEntryStruct
{
    //파일인지 디렉토리인지 구분
    int flag;
    // 파일 이름
    char vcFileName[ FILESYSTEM_MAXFILENAMELENGTH ];
    // 파일의 실제 크기
    DWORD dwFileSize;
    // 파일이 시작하는 클러스터 인덱스
    DWORD dwStartClusterIndex;

    //부모 디렉터리 표시
    char ParentDirectoryPath[FILESYSTEM_MAXFILENAMELENGTH];
    DWORD ParentDirectoryCluserIndex;
} DIRECTORYENTRY;
```

2.4.3 mkdir 명령어 구현

디렉토리 생성하는 명령어를 구현한다. Mkdir folder명을 입력하면 kMakeDirectory 함수가 호출된다. folder명만 파싱하여 opendir 함수를 호출한다. 그 후 kCreateDirectory 함수가 호출된다.

```
static BOOL kCreateDirectory( const char* pcFileName, DIRECTORYENTRY* pstEntry,
        int* piDirectoryEntryIndex )
{
    DWORD dwCluster;

    // 빈 클러스터를 찾아서 할당된 것으로 설정
    dwCluster = kFindFreeCluster();
    if( ( dwCluster == FILESYSTEM_LASTCLUSTER ) ||
        ( kSetClusterLinkData( dwCluster, FILESYSTEM_LASTCLUSTER ) == FALSE ) )
    {
        return FALSE;
    }

    // 빈 디렉터리 엔트리를 검색
    *piDirectoryEntryIndex = kFindFreeDirectoryEntry();
    if( *piDirectoryEntryIndex == -1 )
    {
        // 실패할 경우 할당 받은 클러스터를 반환해야 함
        kSetClusterLinkData( dwCluster, FILESYSTEM_FREECLUSTER );
        return FALSE;
    }

    // 디렉터리 엔트리를 설정
    kMemCpy( pstEntry->vcFileName, pcFileName, kStrLen( pcFileName ) + 1 );
    pstEntry->dwStartClusterIndex = dwCluster;
    pstEntry->dwFileSize = 0;
    pstEntry->flag=1;
    pstEntry->ParentDirectoryPath[0] = '/';
    pstEntry->ParentDirectoryPath[1] = '\0';
    pstEntry->ParentDirectoryCluserIndex = 0;
    //kMemCpy( pstEntry->vcDirectory, "Directory", kStrLen( "Directory" ) + 1);

    // 디렉터리 엔트리를 등록
    if( kSetDirectoryEntryData( *piDirectoryEntryIndex, pstEntry ) == FALSE )
    {
        // 실패할 경우 할당 받은 클러스터를 반환해야 함
        kSetClusterLinkData( dwCluster, FILESYSTEM_FREECLUSTER );
        return FALSE;
    }

    kSetDotInDirectory();

    return TRUE;
}
```

kCreateDirectory 함수는 생성할 디렉토리에게 할당해줄 클러스터를 찾는다. 그 후 현재 디렉토리에서 빈 엔트리를 찾아 할당한다. 마지막으로 생성할 디렉토리의 정보를 stEntry 변수에 저장하고 kSetDirectoryEntryData 함수를 호출하여 디렉토리 엔트리에 저장한다. kSetDirectoryEntryData 함수 안에는 kReadCluster함수가 있는데 파라미터로 루트 디렉토리인 0을 보내는 것을 현재 디렉토리의 클러스터인덱스 값을 보내도록 수정한다.

```
//  디렉터리를 읽음
if( kReadCluster( currentClusterIndex, gs_vbTempBuffer ) == FALSE )
{
    return FALSE;
}
```

2.4.4 cd 명령어 구현

cd folder명을 입력하면 kMoveDirectory 함수가 호출된다. kMoveDirectory 함수 안의 kFindDirectory 함수는 클러스터인덱스로 찾고자 하는 디렉토리의 인덱스 배열을 리턴해준다.

```
DIRECTORYENTRY* kFindDirectory( DWORD currentCluster )
{
    DIRECTORYENTRY* pstEntry = NULL;
    int i;

    // 파일 시스템을 인식하지 못했으면 실패
    if( gs_stFileSystemManager.bMounted == FALSE )
    {
        return NULL;
    }

    // 디렉터리를 읽음
    if( kReadCluster( currentCluster, gs_vbTempBuffer ) == FALSE )
    {
        return NULL;
    }

    // 루트 디렉터리 안에서 루프를 돌면서 빈 엔트리, 즉 시작 클러스터 번호가 0인
    // 엔트리를 검색
    pstEntry = ( DIRECTORYENTRY* ) gs_vbTempBuffer;
    if(pstEntry == NULL)
        return NULL;

    return pstEntry;

}
```

현재 디렉토리의 엔트리들을 리턴받으면 for문을 사용하여 엔트리를 하나하나 돌면서 이동하고자 하는 디렉토리를 찾는다. 비여있지 않고 디렉토리인 파일만을 검색하도록 하여 오류가 없도록 한다. 이동할 디렉토리를 찾으면 이동하기 전의 디렉토리 경로와 클러스터 인덱스를 이동할 디렉토리의 ..(dot-dot) 디렉토리에 저장하여 나중에 cd .. 명령어를 통하여 부모 디렉토리로 이동할 수 있도록 한다. 그 후 이동한 디렉토리의 이름을 path 변수에 더하여 주고 이동한 디렉토리의 클러스터 인덱스를 전역변수인 currentDirectoryClusterIndex에 저장한다.

```
for( int j = 0 ; j < FILESYSTEM_MAXDIRECTORYENTRYCOUNT ; j++ )
{
    if( directoryInfo[ j ].dwStartClusterIndex != 0 &&
     kMemCmp(directoryInfo[ j ].vcFileName,vcFileName,kStrLen(vcFileName))==0 &&
     directoryInfo[j].flag ==1)
    {
        kMemCpy(temp_path,path,kStrLen(path)+1);
        temp_index = currentDirectoryClusterIndex;

        if(kMemCmp(path,"/",2)==0)
            kMemCpy(path + kStrLen(path),vcFileName,kStrLen(vcFileName)+1);
        else{
            kMemCpy(path + kStrLen(path),"/",1);
            kMemCpy(path + kStrLen(path),vcFileName,kStrLen(vcFileName)+1);
        }

        currentDirectoryClusterIndex = directoryInfo[ j ].dwStartClusterIndex;
        kSetClusterIndex(currentDirectoryClusterIndex);
        directoryInfo = NULL;
        directoryInfo = kFindDirectory(currentDirectoryClusterIndex);
        if( directoryInfo[ 0 ].dwStartClusterIndex != -1 )
        {
           kSetDotInDirectory();
            kUpdateDirectory(0,".",path,currentDirectoryClusterIndex);

        }

        directoryInfo[1].ParentDirectoryCluserIndex = temp_index;
        kMemCpy(directoryInfo[1].ParentDirectoryPath,temp_path,kStrLen(temp_path)+1);
        kUpdateDirectory( 1,"..",temp_path, temp_index );

        break;

    }
```

cd . or cd ..을 하면 기존에 저장하였던 정보를 전역변수에 다시 저장하여 현재 디렉토리, 부모 디렉토리로 이동할 수 있도록 한다.

```
if(kMemCmp(vcFileName,".",2)==0){
    currentDirectoryClusterIndex = directoryInfo[0].ParentDirectoryCluserIndex;
    kSetClusterIndex(currentDirectoryClusterIndex);
    kMemCpy(path,directoryInfo[0].ParentDirectoryPath,kStrLen(directoryInfo[0].ParentDirectoryPath)+1);
}
else if(kMemCmp(vcFileName,"..",3)==0){

    currentDirectoryClusterIndex = directoryInfo[1].ParentDirectoryCluserIndex;
    kSetClusterIndex(currentDirectoryClusterIndex);

    kMemCpy(path,directoryInfo[1].ParentDirectoryPath,kStrLen(directoryInfo[1].ParentDirectoryPath)+1);
}
```

2.4.5 rmdir 명령어 구현

rmdir folder명을 입력하면 kRemoveDirectory 함수가 호출된다. 그 후 remove 함수를 사용하여 해당 디렉토리를 제거한다.

```c
static void kRemoveDirectory( const char* pcParameterBuffer )
{
    PARAMETERLIST stList;
    char vcFileName[ 50 ];
    int iLength;

    // 파라미터 리스트를 초기화하여 파일 이름을 추출
    kInitializeParameter( &stList, pcParameterBuffer );
    iLength = kGetNextParameter( &stList, vcFileName );
    vcFileName[ iLength ] = '\0';
    if( ( iLength > ( FILESYSTEM_MAXFILENAMELENGTH - 1 ) ) || ( iLength == 0 ) )
    {
        kPrintf( "Too Long or Too Short File Name\n" );
        return ;
    }

    if( remove( vcFileName ) != 0 )
    {
        kPrintf( "File Not Found or File Opened\n" );
        return ;
    }

    kPrintf( "File Delete Success\n" );
}
```

remove함수에서는 다양한 검사를 한다.파일의 이름이 제대로 되어있는 검사, 파일이 존재하는지 검사, 파일이 닫혀 있는지 검사한 후에 모두 만족을 하면 디렉토리를 구성하는 클러스터를 모두 해제하고 디렉토리 엔트리를 빈 것으로 설정한다.

2.4.6 구동 확인

```
MINT64:/>dir
    .                           Directory
    ..                          Directory
MINT64:/>mkdir temp
Directory Create Success
MINT64:/>createfile a.txt
File Create Success
MINT64:/>dir
    .                           Directory
    ..                          Directory
    temp                        Directory
    a.txt                       0 Byte                  0x2 Cluster
```

```
MINT64:/>cd temp
MINT64:/temp>dir
    .                           Directory
    ..                          Directory
MINT64:/temp>createfile b.txt
File Create Success
MINT64:/temp>dir
    .                           Directory
    ..                          Directory
    b.txt                       0 Byte                  0x3 Cluster
MINT64:/temp>_
```

```
MINT64:/temp>createfile b.txt
File Create Success
MINT64:/temp>dir
    .                           Directory
    ..                          Directory
    b.txt                       0 Byte                  0x3 Cluster
MINT64:/temp>cd .
MINT64:/temp>cd ..
MINT64:/>dir
    .                           Directory
    ..                          Directory
    temp                        Directory
    a.txt                       0 Byte                  0x2 Cluster
MINT64:/>cd .
MINT64:/>cd ..
MINT64:/>rmdir temp
File Delete Success
MINT64:/>dir
    .                           Directory
    ..                          Directory
    a.txt                       0 Byte                  0x2 Cluster
MINT64:/>
```

2.4.7 개발 시 경험한 문제점 및 해결점

1. 파일시스템 구조 인식의 어려움

이론 수업시간에 배웠던 파일시스템과 구현하고자 하는 MINT OS의 파일시스템이 구조적으로 달랐기 때문에 이해하는 과정이 오래 걸렸다. 또한 단일 디렉토리 파일시스템에서 서브 디렉토리 파일시스템으로 바꾸는 과정이 많은 시간을 소요하게 하였다.

2. 서브 디렉토리 구조로 인한 명령어 구조 변화

기존의 명령어들은 모두 루트 디렉토리 안에서 수행되다고 가정하에 만들어졌다. 하지만 서브 디렉토리 구조로 수정한 후에는 다양한 디렉토리에서 파일 생성과 디렉토리 생성, 디렉토리 보는 명령어 등 모든 명령어를 현재 디렉토리에서 수행할 수 있도록 수정해야 했다.

3. 디렉토리 안의 dot, dot-dot 구현

디렉토리 안의 .(dot), ..(dot-dot) 디렉토리를 구현하기 위해서는 해당 디렉토리마다 이동할 수 있는 정보를 가지고 있어야했다. 그런 정보를 언제 입력해야하는지에 대한 문제가 발생하였다. 그래서 처음 디렉토리를 만들 때와 해당 디렉토리로 이동할 때 정보를 입력했다.

4. 디렉토리 엔트리를 수정하였을 때 갱신되지 않았던 문제

구현하다 보니 디렉토리 엔트리를 수정해야하는 상황이 발생하였다. 하지만 엔트리를 수정하였지만 반영되지 않은 상황이 나타났다. 엔트리 수정 후 업데이트 해주지 않았기 때문이었다. 해당 문제를 해결하는데 많은 시간이 소요되었다.

3. 조원별 기여도

신정은: 33.3%
조재호: 33.3%
조한주: 33.3%