**Project 4 Design Report**

성해빈, 송창근

**1. 시스템 분석**

1. inode

inode는 유닉스 계통 파일 시스템에서 사용하는 자료구조이다. 각 파일은 하나의 inode를 가지고 있으며, 각기 고유한 inode number를 통해 구분된다. inode에는 일반적으로 소유자 목록, 접근 정보, 파일 형태 등 해당 파일에 대한 정보가 존재하며, 핀토스에서는 inode.c에 정의되어 있다. linux의 file system 구조는 fd->open file table -> vnode 이고, pintos의 file system 구조를 보면 fd->inode이다. inode는 모든 프로세스가 공유하는, 가장 low 한 레벨에 있는 자료구조이며, disk와 밀접한 연관이 있다. 서로 다른 fd라도 같은 inode를 지목할 수 있다.

1. new system call

이번에는 파일과 관련된 system call을 추가할 계획이다.

chdir : cd로도 많이 쓰며, 현재 작업 디렉토리를 바꾼다. 절대와 상대 경로 둘 다 지원

mkdir : 디렉토리를 생성한다. 상위 디렉토리는 무조건 있어야하고 만드려는 디렉토리는 무조건 없어야 성공한다. 절대, 상대 지원

readdir : fd로 받은 디렉토리에 있는 엔트리를 전부 char \* name에다 저장해준다.

isdir : fd가 디렉토리인지 아닌지 구분한다.

inumber : fd에 연결되어있는 inode 를 리턴한다.

**2. 문제 해결 (구현)**

1. **Buffer Cache** (10 points)

디스크에 접근하는 것은 캐시나 램에 접근하는 것에 비해 매우 느리기에 파일을 관리할 때 디스크에서 직접 작업하는 것은 매우 비효율적이다. 따라서 파일을 관리할 때는 사용할 디스크 블록을 메모리에 캐시하여 작업하고 작업이 끝나면 그 내용을 디스크에 반영하는 것이 효율적이다.

**현재 구현** – 바운스 버퍼를 사용한다. 이는 malloc()을 통해 디스크의 섹터 인터페이스를 시스템 호출의 바이트 인터페이스로 변환하여 할당하는 방법이다.

**필요한 구현** – 바운스 버퍼를 제거하고 섹터에서 직접 버퍼 캐시로 복사하는 밥법을 구현해야 한다. 버퍼 캐시에는 다음과 같은 기능이 포함되어야 한다.

* 캐시 교체 알고리즘: 캐시에 파일 블록들을 어떤 방식으로 교체할지 결정해야 한다. 우리는 Clock 알고리즘을 사용할 것이다.
* Write-Back: 파일이 수정되면, 우선 디스크에 반영하지 말고 더티 블록을 유지하다가 해당 블록이 evict될 때 디스크에 수정 사항을 반영해야 한다.  
  파일을 쓸 때 충돌이 일어나면 파일 시스템이 불안정해지므로 주기적으로 모든 더티 블록들을 디스크에 반영해야 한다.
* Read-Ahead: 파일의 한 블록을 읽을 때, 자동으로 그 다음 블록을 캐시로 가져와야 한다. 이 기능은 백그라운드에서 실행되어야 한다.
* 핀토스가 중지되었을 때, filesys\_done( )에 의해 파일 시스템이 전부 정리된다. 따라서 이때 캐시 블록들의 내용이 디스크에 반영되어야 한다.

캐시는 64섹터 미만의 크기로 제한된다.

1. **Indexed and Extensible Files** (10 points)

대부분의 운영체제에서는 블록이 단편화 되어있어도 파일을 분산하여 저장할 수 있으며, 파일 크기 또한 동적으로 정해진다. 이들을 포함한 파일 시스템을 구현해야 한다.

**현재 구현** – 파일이 단일 범위의 연속된 블록으로만 할당될 수 있어 외부 단편화 상황에서는 충분한 개수의 블록이 비어있다고 해도 할당하지 못할 수 있다. 또한 파일 크기가 파일을 생성할 때 주어진 크기로 제한된다.

**필요한 구현** –inode 구조를 수정하여 외부 단편화 문제를 해결한다. 이를 위해서는 직접, 간접, 이중 간접 블록을 활용한 새 인덱스 구조를 구현해야 한다. 이는 inode에서 inode\_disk의 list를 만들어 관리할 것이다. 우리는 현재 extent 방식으로 돼있는 inode structure 구현을 수정할텐데, external fragmentation 이 발생하지 않기 위해 index 방식의 doubly indirect block을 이용할 계획이다. external fragmentation은 할당된 블록이 띄엄띄엄 있어서 연속된 메모리가 없어 할당이 안 되는 경우다. doubly indirect block방식으로 하면 8MB 까지 지원할 수 있어서 거기에도 장점이 있다. (일단 8MB 이상은 지원하지 않는다고 핀토스 매뉴얼에서 못 박았기 때문에)   
이 방식의 좋은 점은 또 file growth를 지원할 수 있다는 점이다. 우리는 파일의 크기가 동적으로 확장될 수 있도록 구현해야 한다. 대부분의 파일 시스템에서는 파일을 처음 생성하면 크기가 0인 파일이 생성되고, 파일의 끝에서 쓰기가 수행될 때마다 파일이 확장된다. 이와 같은 기능을 구현해야 하며, 루트 디렉토리 파일에서도 똑같이 적용된다. 아래는 함수들이 EOF 관련하여 어떻게 동작하는지에 대한 조건이다.

* File Seek: 유저 프로그램이 현재 EOF를 지나 탐색할 수 있다.
* File Read: EOF를 지난 위치를 읽으면 바이트를 반환하지 않는다.
* File Write: EOF를 지난 위치에 쓰면 파일이 그 크기까지 확장되며, 지난 EOF와 현재 쓰는 위치 사이는 전부 0으로 초기화되어야 한다.

파일 시스템은 (파티션 크기 – 메타데이터 공간)만한 파일 크기를 지원해야 한다.

각 inode는 디스크 섹터 하나 안에 저장된다. 따라서 inode가 가질 수 있는 블록 포인터의 개수가 제한된다. (섹터는 512바이트)

1. **Subdirectories** (10 points)

서브디렉토리를 가질 수 있도록 파일 시스템을 수정한다.

**현재 구현** – 파일 이름에 14자 제한이 존재하며, 디렉토리가 디렉토리를 포함할 수 없고 오직 파일만을 포함할 수 있다. 즉 네임 스페이스에 계층 구조가 존재하지 않는다.

**필요한 구현** –디렉토리가 파일뿐 아니라 다른 디렉토리를 포함할 수 있어야 한다. 파일들과 마찬가지로 디렉토리 또한 확장이 가능해야 하며, 파일의 전체 경로를 14자 이상으로 표현할 수 있게 할 것이다.(이는 선택사항, READDIR\_MAX\_LEN을 수정) 파일 시스템은 다음과 같은 조건을 만족해야 한다.

* 디렉토리 분리 문자는 ‘/’이다. ‘.’ 이나 ‘..’ 등도 유닉스와 같은 기능을 포함해야 한다.
* 상대 경로와 절대 경로를 다 지원해야 한다. 우리는 이 파싱을 지원하기 위해 앞에서부터 차례대로 while문을 돌면서 case 문으로 경우를 나눌 생각이다.
* 각 프로세스마다 별도로 현재 디렉토리 정보를 관리해야 한다. 하위 프로세스는 상위 프로세스의 현재 디렉토리 정보를 상속받아 시작한다.

시스템 호출에서는 open, close, remove가 디렉토리에서도 정상적으로 작동하도록 수정한다. remove는 루트 디렉토리를 제외한 빈 디렉토리를 삭제할 수 있어야 한다. 또한 새로운 시스템 호출을 구현해야 한다.

* bool chdir(const char \* dir): 프로세스의 현재 디렉토리를 dir로 변경한다. 성공하면 true, 실패하면 fail을 반환한다.
* bool mkdir (const char \* dir): dir이라는 디렉토리를 생성한다. 성공하면 true, 실패하면 fail을 반환한다. mkdir ("/a/b/c")라는 명령이 있다면 /a/b가 존재하고 /a/b/c가 존재하지 않는 경우에만 성공한다.
* bool readdir (int fd, char \* name): 디렉토리 엔트리를 하나씩 읽어온다. 먼저 fd가 디렉토리를 나타내는지 확인한 뒤, fd에서 디렉토리 엔트리를 읽어 파일 이름을 char\* name에 저장하고 true를 반환한다. 읽어오는 과정 중에 문제가 있으면 오류를 실행해야 한다. 더 이상 디렉토리에 항목이 남아있지 않으면 false를 반환한다.
* bool isdir (int fd): fd가 디렉토리의 fd일 경우 true를 반환하고 그렇지 않으면 false를 반환한다.
* int inumber (int fd): fd에 해당하는 inode의 inode number를 반환한다. 파일이나 디렉토리에서 전부 실행되어야 하며, 이 inumber는 파일이 존재하는 동안 유일해야 한다.

1. **Synchronization** (5 points)

여러 프로세스가 같은 캐시나 파일, 디렉토리를 다룰 때는 여러 규칙이 필요하다. 그런 규칙이 없다면 충돌이 일어나거나 잘못된 메모리 참조로 인해 파일이 손상되는 등의 문제가 생길 수 있기 때문이다. 다양한 상황에 맞는 스케줄링 규칙을 구현한다.

**현재 구현** – 동기화 규칙이 존재하지 않는다.

**필요한 구현** –동기화를 위해서는 함수들을 수정하고 동기화를 위한 새로운 자료구조를 만들어애 한다. 동기화 규칙은 아래와 같다.

* 서로 다른 캐시 블록에 대한 작업은 독립적으로 진행되어야 한다. 특정 블록에서 I/O를 할 때, 다른 블록에 대한 작업은 이를 기다리지 않고 동시에 진행되어야 한다.
* 같은 파일을 여러 번 읽을 때, 이는 서로를 기다리지 않고 동시에 진행되어야 한다.
* 파일 크기가 확장되지 않는 경우, 여러 프로세스에서 같은 파일에 대한 쓰기 작업을 동시에 진행할 수 있다.
* 파일 크기가 확장되어 새로운 섹션에 데이터를 쓰는 경우,각 쓰기 작업은 원자적으로 처리되어야 한다.
* 동일한 파일에 쓰기 작업을 진행하는 프로세스의 경우, 다양한 결과가 도출될 수 있다. 예를 들어, 프로세스 A는 프로세스 B의 이전 결과를 덮어 쓸 수 있고, 그 반대도 마찬가지로 가능하다.
* 다른 디렉토리에 대한 작업은 동시에 수행되어야 한다. 같은 디렉토리에 대한 작업은 다른 작업이 끝날 때까지 서로 기다릴 수 있다.

**3. Conclusion**

마지막 핀토스 프로젝트는 지금까지 프로젝트 2와 3에서 자세한 것은 모른 채 그저 사용해 왔던 파일 시스템을 구현하는 것이다. 현재 핀토스의 파일 시스템은 우리가 흔히 생각하는 방식과는 많이 다른데, 이번 프로젝트를 구현 완료한다면 핀토스의 파일 시스템이 우리가 흔히 사용하고 있는 파일 시스템같이 편리하게 바뀔 것이다. 버퍼 캐시와 동기화를 구현하는 것이 기대된다.