

Pintos Project 5: Filesystem

담당 교수: 박성용

학번 / 이름: 20171643, 박준혁

개발 기간: 12/07–12/25

1. 개발 목표

2022년도 가을학기 운영체제 과목의 대망의 마지막 Project Phase인 pintOS 'File System'에선 본 과목 강의 후반부에 학습했던 **File Control Block, Directory Service** 등을 위시한 File System 개념들을 실제 pintOS 상에 접목시킴으로써 현대 OS에서 Disk 및 File을 어떻게 관리하는지, 그리고 Disk와 Main Memory를 어떻게 연동시키는지, 나아가 그 연동 과정에서 Buffer Cache와 같은 속도 향상 Method를 어떻게 구현하는지 등을 다각도로 확인하고 체험해볼 수 있다.

지난 Phase 4까지의 pintOS File System엔 몇 가지 한계점이 존재했다. Project 안내 PDF에서도 명시되어 있듯, 기존 File System에선 inode(FCB, File Control Block)가 하나의 Sector(Block)만 점유할 수 있고, 그에 따라 가질 수 있는 Block Pointer의 개수가 한정될 수 밖에 없었다. 또한 그 과정에서 해당 inode가 가리키는 Data에 대한 Block들을 Linear하게 Disk에 저장하기 때문에 필연적으로 External Fragmentation이 극심해질 수 밖에 없었다.

한편, Subdirectory를 가지지 못한다는 점도 문제였다. 기존 File System에선 오로지 하나의 Root Directory만 두어 File들을 관리하고 가리켰는데, 본 Phase에선 우리가 익히 알고 있는 실제 Directory 구조를 pintOS File System이 관리할 수 있도록 만들어주는 것이 필요하다.

그리고 Manual과 Project 안내 PDF의 내용대로 Buffer Cache도 구현한다. Buffer Cache를 구현함으로써 '매번 Disk I/O가 발생할 때마다 즉각적으로 처리하는 것'의 비효율성을 제거하도록 한다. 쉽게 말해, Disk와 Main Memory가 소통할 때, 매번 I/O를 진행하지 않고, 모아두었다가 필요할 때만 '실제 I/O'를 수행하는 것이다.

이를 위해 우리는 **inode Structure**를 대대적으로 수정하여 **Extensible File, File Growth를 가능케** 해야 한다. 이어 **Directory Service**를 수정하여 **Subdirectory**가 제공될 수 있도록 만든다. 또한 그 과정에서 **Buffer Cache**를 마련해 **효율적인 I/O가 가능하게** 만들어 준다. 이때, inode, Buffer Cache와 같은 Shared Resource에 대한 Race가 펼쳐질 수 있고, 이를 Synchronize하는 작업도 필요할 것이다. 이렇게 Project 5를 진행한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Extensible file & file growth

기존 pintOS File System에서 File Size가 고정되어 있던 이유는 무엇인가? 그렇다. inode Structure가 연장될 수 없는, Linear한 Format을 가졌기 때문이었다. 따라서 우리는 이러한 **inode의 구조를 변경해야 한다**. 알다시피, inode의 확장은 결국 Indirect Pointer를 두어 해결할 수 있다. **Multi-Level의 Indirect Pointer**를 두면 더더욱 효율적으로 확장할 수 있음도 명확하다. 따라서 우리는 Indirect Pointer를 둘 것이다. 이때, 어떤 구조로 두느냐는 **Design-Dependent**한데, 본인은 아래와 같은 구성을 지닌 16개의 Pointer들을 두기로 결정했다.

'16 Pointers = 5 Direct Pointers + 10 Indirect Pointers + 1 Double Indirect Pointer'

이렇게 정한 데에는 사실 그렇게 큰 이유는 없다. 본 강의 시간에서 다루었던 것처럼 inode Structure에 Pointer 구성이 어떻게 되느냐는 해당 File System에서 머금을 수 있는 최대의 File Size를 의미하는데, 위와 같은 구성을 갖추게 설계하면 우리의 pintOS는 최대 $(5 + 10 \times 2^7 + 1 \times 2^{14}) \times 2^9 = 2^{23}$ 에서 2^{24} 사이의 값, 즉, 8MB 이상의 크기'의 File까지 가질 수 있긴 하다. 그러나, pintOS Manual에선 특별히 File Size에 제한을 두지 않았고, 동시에 8MB를 초과하는 File 을 할당하는 Test Case도 없다. 따라서, 별 다른 의미는 없다. 단지, Indirect를 도입함으로써 External Fragmentation을 피할 수 있는 구조를 갖추었다는 것이 중요할 뿐이다.

이렇게 inode Pointing 구조를 갖추게 되면, 이러한 inode가 On-Disk에서 In-Memory로 매칭될 때 추가적으로 필요한 정보들을 명시한 후, 이어 inode와 관련한 각종 기반 Procedure들을 수정해 File Extension, File Growth가 정상적으로 이루어질 수 있도록 System을 수정한다. inode와 관련한 거의 모든 기능을 수정해야 하기에 대대적인 작업이 필요하다. 자세한 Detail은 아래서 서술한다.

이러한 **inode Structure 변경을 통해 우리는 File이 Disk에 Linear하게 배치되어 발생했던 External Fragmentation, File Size 확장 불가 문제를 해결할 수 있게 된다**. 참고로, inode를 Disk와 Memory 사이에서 옮기는 과정엔 Buffer Cache 가 관여할 것임을 미리 염두하자.

2. Subdirectory

앞서 언급한 바와 같이, 기존의 pintOS File System에선 오로지 하나의 Root Directory만 제공해 사실상 모든 File이 Root Directory 내에 상주하도록 했었다. 우리는 본 Phase에서 Subdirectory가 가능하게 하여 이 문제를 해소한다. 이는 굉장히 간단히 해결할 수 있다.

"Directory Entry가 File뿐만 아니라 또 다른 Directory를 가리킬 수 있게 하자."

물론 그 구현 Detail은 복잡하긴 하나, 원리는 간단하다. Directory Entry 및 관련함수들을 수정하기만 하면 된다. 이때, 실제 Linux에서 Directory 관리 시 사용하는 **Convention**을 지켜준다. 이를테면 Directory 내에 하부 File 및 Directory가 없어야 해당 Directory를 삭제할 수 있다든지, 또는 Open된 Directory는 삭제되어 선 안 된다든지 하는 식의 조건들을 구현하도록 한다.

가장 중요한 것은 **Persistency**를 만족시키는 것으로, pintOS가 가상 환경에서 종료 직후 바로 다시 부팅되었을 때 데이터가 유지되고 있느냐를 본 Phase Test Case들에서 중점적으로 검사될 것이다.

이러한 각 종 조건을 만족시키기 위해 우리는 pintOS 상에서 Directory Service를 담당하는 코드 부분을 위의 inode 때와 같이 대폭 수정해야 하며, 동시에, 해당 Directory 기능을 사용자에게 제공하는 'Directory 관련 System Call들'도 구현해야 한다. 이를 통해 우리의 pintOS는 Subdirectory 기능을 정상적으로 제공할 수 있게 된다.

3. Buffer cache

우리는 Buffer Cache를 구현한다. 상기한 것처럼, Disk와 Main Memory가 소통할 때, I/O 요청이 있을 때마다 매번 일일이 Disk I/O를 수행하는 것은 굉장히 소모적으로 비효율적인 행위이다. Disk의 접근 속도가 Memory에 비해 현저히 느리기 때문이다.

우리는 이를 **Buffer Cache** 개념으로 개선하는 것이다. 만약 I/O의 대상이 되는 Data가 Cache에 존재한다면, 굳이 Disk에 접근하지 말고 Cache에서 바로 꺼내 사용하자는 것이다. Cache에 없는 Data만 Disk에서 뽑아 쓰자는 것이다. 우리는 64개의 Sector(Block)로 이루어진 Buffer Cache를 운용하도록 한다.

한편, Buffer Cache의 구현은 사실 새롭지 않은 내용이다. Buffer Cache의 구현 방법은 다양한 Variation이 존재하는데, 그 중 가장 대중적이고 쉬운 방법은 바로 **Clock Algorithm**을 사용하는 것이다. Clock Algorithm, 그것은 우리가 바로 직전 Phase에서 Page Eviction 구현 시 사용했던 개념으로, 상당히 친숙한 개념이다. 그 당시 사용했던 논리를 그대로 적용하면 된다. Buffer Cache가 꽉 차면, Buffer Cache를 Circular하게 순회하며 'Un-accessed Slot'을 찾고, 해당 Slot을 Evict한다. 이때, 그러한 순회 과정에서 'Un-accessed Slot'이 나오기 전까지 등장하는 모든 'Accessed Slot'에겐 Second Chance를 부여하는 것이다. 어렵지 않게 구현할 수 있을 것이다.

오히려 이때 신경 써야 할 부분은 OS 종료 혹은 **File System Unmount**, **Termination** 시의 처리로, 이러한 상황에서 Buffer Cache의 내용은 Disk에 Update되어야 한다. 이때, 모조리 Update(Flush)할까? 당연히 아닐 것이다. 굳이 그럴 필요 없이, **Dirty한 Slot의 Data만 Update**하면 되겠다. 이 역시 직관적으로 이해할 수 있을 것이다.

우리는 이러한 Buffer Cache 개발을 통해 보다 효율적인 File System 운용을 실현할 수 있게 된다.

B. 개발 내용

- **Extensible file & File growth**

- ✓ **Index structure와 Management**

우리가 Project Phase 5에서 구축하는 File System에선, 모든 File에 대한 작업은 inode Structure를 거쳐야 한다. 이는 앞서 말한 바와 같이, **inode Structure를 정의하는 것에서부터 시작한다.** 16개의 Pointer를 두고, 그 중 5개는 Direct Pointer, 10개는 Indirect Pointer, 1개는 Double Indirect Pointer라고 했다. 이 밖에 On-Disk inode에는 File의 Size, Magic Number 등의 정보가 기록되어야 한다. 본인은 여기에, 해당 inode가 Directory를 이루는 inode 인지를 나타내는 Flag 변수도 추가하도록 하겠다. 이렇게 해서 'On-Disk inode'를 마련할 수 있다.

이제 이를 토대로 inode가 In-Memory에서 바라볼 때 필요한 Structure도 구현한다. 위의 'On-Disk Structure'에 몇 가지 추가 정보를 기입하면 되는데, 대표적으로 Reference Count 값, Deny Write Count 값, Read Bytes 정보 등이 있겠다.

이렇게 해서 **inode Structure가 완성되면**, 이제 우리는 이를 토대로 **inode Management**을 수행한다. inode와 관련한 대표 Operation들을 기준으로 설명하겠다.

1) inode 생성 및 해제: inode Creation(Allocation) 및 Closing(Deallocation) 과정은 사실 그렇게 복잡한 내용은 아니다. 기존 System과 같이 공간 할당 및 데이터 복사를 하되, 새롭게 바뀐 구조대로 해주면 되는 것이다. 즉, 코드 단위로 보면, 구조체의 필드들에 맞게 값을 설정하고 초기화하면 된다.

복잡하고 어려운 부분은 바로 그러한 **inode에 대한 Free Map 할당 시 inode의 Index Structure와 현재 Given File Size를 고려해서 Block 할당을 수행해야 한다는** 점이다. 만약, inode의 생성 및 변형 과정에서 해당 inode와 맵핑된 File의 Size가 '현재 inode가 가리키는 사이즈 범위'를 넘어서면 그 넘어서는 만큼을 각 Index Structure에 알맞게 Direct Pointer부터 순서대로 고려하여 Free Map 공간을 할당하는 것이다. Indirect Pointer의 경우, 해당 Pointer가 가리키는 Block 내의 최대 가능 Pointer 개수 ($128 = 512/4$) 만큼을 일일이 살펴보며 할당을 진행하면 되겠다.

Double Indirect의 경우도 결코 다르지 않다. 역시나 그대로 훑으면서 확장해야 하는 만큼 Free Map을 할당하고 Block Write을 진행한다. 단지, 이 과정이 Indirect에선 Linear Traversal 형태로 이뤄진다면, Double에선 128 x 128번의 Nested Traversal 형태로 이뤄진다는 점이 다를 뿐이다.

이렇게 해서 inode가 특정 가변 사이즈 File을 가리킬 수 있게 된다. inode 생성 시 말이다. 자, 그렇다면 inode 해제(Close) 시엔 어떠할까? 똑같다. 단지, Free Map 공간을 할당하고 Disk Write을 하던 것을 Free Map 공간 해제 및 Disk Read하는 것으로 바꾸면 된다. 그 동작 구조나 Flow는 결코 다르지 않다.

2) **inode Read/Write**: 그리고, 그러한 inode의 생성 및 해제 사이에서, inode가 가리키는 File에 대한 Read/Write I/O가 이뤄질 때, File의 Size가 변화해야 하는 지점이 생긴다면, 위의 생성 시 설명한 논리와 똑 같은 논리로 inode Pointer 구조를 건드려 File Extension을 수행하면 된다.

물론, inode, Index Structure에 대한 설명이 이렇게 글로만 표현하면 간단하다. 그러나, 이후 아래에서 설명하겠지만, 이러한 Management 과정을 Code-Level에서 바라보면 굉장히 코드 양이 많고 복잡하다. 그러나, 그 논리 구조는 위에서 설명한 그대로이기 때문에, 이 점을 기억하도록 하자. 아무튼, **이렇게 해서 Extensible File, File Growth를 구현할 수 있게 된다.**

- **Subdirectory**

- ✓ **Directory entry 관리 방법**

inode에 대해 알아보았다. 그러한 FCB(File Control Block)은 두 가지 객체가 가리킬 수 있다. 하나는 Open File Structure이다. 쉽게 말해 Open File Descriptor, 즉, Process 입장에서 특정 File을 열었을 때, 그 File과 소통하기 위한 데이터 구조를 의미한다. 그러한 구조는 inode를 가리켜 실제 File의 Data 및 Metadata를 접근한다.

한편, 또 다른 하나의 객체는 바로 Directory이다. Directory 내부엔 File이 있기 마련이고, 그 내부 File(또는 Directory 그 자체)을 가리킬 때 바로 그 inode를 Point한다. pintOS에선 바로 이러한 구조를 'filesys/directory.h'에서 관리하는데, 그 내부의 두 가지 Pre-defined 구조체를 주목한다. 하나는 'dir' 구조체이다. 특정 Directory가 있다면, 그 Directory 자체에 대한 inode 및 위치를 기록한다. 이어, 또 다른 구조체는 바로 'dir_entry', 즉, Directory Entry 구조체이다. 이 구조체에는 해당 Entry가 가리키는 Sector Number와 해당 File의 이름 등이 담겨있다.

자, 이렇게 Directory 관련 Data Structure를 알았으니, 본격적으로 관리 형태를 알아보자. 먼저, Directory를 생성한다. Directory는 'Special Type of File' 이므로, 마찬가지로 그대로 Corresponding inode를 생성한다. 다만, 앞선 항목에서 언급했던 것처럼, 해당 inode가 Directory를 위한 inode임을 특정

Flag로 명시하도록 한다. 참고로 이렇게 Directory를 생성할 때, 해당 inode가 가리키는 File의 Size는 Directory Entry의 개수에 Directory Entry Size를 곱한 Size가 된다.

Directory의 생성이 완료되면, 그 Corresponding inode에 대한 각 종 연산들을 정의한다. Open이라 하면, 'dir' 변수를 할당하고, 해당 변수가 해당 inode를 가리키게 하는 식으로 말이다. 이러한 대부분의 기능은 이미 pintOS에서 제공한다. 우리가 본 Phase에서 수정해야 하는 부분은 다음과 같은 부분만 해당한다.

- **inode Structure에 '또 다른 inode와의 Parent-Child 관계를 명시할 수 있는 Parent Pointer'를 둔다.**
 - 이때, '또 다른 inode'라 함은, Directory에 포함될 수 있는 File 또는 Subdirectory를 나타내는 inode가 되는 것이다.
 - ◆ 그리고 System에 Directory 연산 시 이러한 관계를 고려해서 연결 관계 Tree가 생길 수 있도록 만들어준다. 즉, Directory Addition 및 관련 System Call을 구축하면 된다.

이렇게 간단히 Directory Entry 및 inode Structure를 조정해 Subdirectory 기능을 구현할 수 있게 된다. 다만, 이러한 Directory Service를 실제 User Program에서 문제 없이 사용할 수 있게 **System Call을 구축하는 부분은 까다롭다**. File System에 새로운 File을 Create할 때, 또는 특정 File을 Open할 때의 기존의 처리 방식을 모조리 수정해야 한다. 기준 pintOS에서는 모든 File이 동일 Root Directory에 존재했기 때문에 File의 생성 및 Open 과정에서 File Path를 전혀 고려하지 않아도 됐었는데, Subdirectory 기능이 구현된 이상, 각 File의 위치 관계를 정확히 알아야 하기 때문이다.

이는 Given File이 위치한 Directory, 해당 File의 Name을 적절히 조정함으로써 처리할 수 있다. 사용자로부터 넘겨 받은 Name에 명시된 경로가 절대 경로인지, 상대 경로인지를 따져 각 Case에 대해 알맞은 String 처리를 하는 것이 핵심이다. 예를 들어, "/sogang/junhyeok.exe"라는 File을 생성하는 상황이라면, 이 경로는 절대 경로 방식이므로 먼저 Root Directory를 Open한다.

이어, 거기서 매 '/' 문자를 기준으로 Token Cutting을 하며 Parent-Child 관계를 Corresponding inode들에 명시한다. sogang Directory의 inode의 Parent

를 Root Directory로 지정하고, junhyeok.exe의 inode의 Parent를 sogang Directory로 지정하는 식으로, String 처리와, 그 String이 가리키는 inode를 찾아 그 inode들 간의 관계를 만들어주고, 그러한 inode 관계 Tree가 구축이 완료된다면, 그제서야 상황의 목적(예를 들면 File의 생성)을 달성하는 식으로 File-Related System Call들을 구현한다.

- **Buffer cache**

- ✓ **Buffer cache eviction 방식**

앞서 '개발 범위' 항목에서 언급한 것처럼, Buffer Cache의 구현은 현 시점에서 굉장히 친숙하다. 지난 Phase 4에서 구현했던 **Clock (Second Chance) Algorithm**을 그대로 구현하면 된다. 단지 그 Shared Buffer의 Slot Structure가 Disk Sector(Block), 그리고 Reference Count, Free/Access/Dirty 여부 등을 담으면 된다는 점만 유의하면 된다.

자, 그러한 Buffer가 마련되면, **Iterator가 Buffer를 Circular Way로 순회하면서 Un-Accessed Slot을 찾는** 것이다. First-Fit 논리로 찾으며, 그 찾는 과정에서 마주치는 모든 Accessed Slot에겐 Second Chance를 부여한다.

Un-Accessed Slot을 찾게 되면, 해당 Slot이 가리키는 Sector를 Buffer에서 Evict한다. 이때, 그 Slot이 Dirty하다면, Disk에 그 Modified Data를 반영하면 되는 것이다. 간단히 이해할 수 있다.

이러한 Eviction은 Buffer Cache가 필요한 순간, Cache가 꽉 차 있으면 이뤄질 수 있도록 설계하고, **File I/O, 즉, inode Read/Write 시에 Data를 Caching함으로써 사용할 수 있다.** 참고로, Buffer Cache는 Shared Resource이기 때문에 Operation 수행 과정에서 Synchronization을 적용해야 한다.

- ✓ **Buffer cache flush 방식**

이러한 Buffer Cache의 운용에 있어서 Flush와 Write Behind의 고려는 가장 중요한 부분 중 하나이다. Disk와 Main Memory 사이에 Buffer Cache를 운용하다가 OS가 갑자기 종료되거나, 해당 File System이 Unmount된다고 해보자. 만약, Buffer Cache의 Data에 아무런 처리를 하지 않고 그대로 System이 끝

마친다면 System의 Persistency는 유지되지 못할 것이다.

이러한 Persistency를 보장하기 위해 우리는 Flush 기능을 추가해야 한다. 이는 File System이 종료되는 시점에 전체 Buffer Cache에 대한 Traversal이 이뤄지게 해, Dirty Slot의 Block은 모조리 Disk에 Update할 수 있도록 처리해 주면 된다. 아래 Code-Level Detail에서 이 '시점'을 보다 자세히 확인하자.

이렇게 Buffer Cache를 운용할 수 있다.

3. 추진 일정 및 개발 방법

A. 추진 일정

이번 Project 5는 불가피하게 시험 기간에 수행되어야 하기 때문에 본인은 해당 Project가 나오자 마자 바로 개발을 시작했다. 안내 강의 및 Manual, 각종 인터넷 자료를 토대로 이론 학습을 마친 후 든 생각은 본 Project가 굉장히 까다롭고 시간이 오래 걸릴, 난이도가 높은 Project라는 점이었다. 따라서 12월 초부터 바로 개발을 진행했으며 시험 기간 1주를 제외하고 약 실제 개발은 약 2주 정도 소요되었다. 구체적인 과정은 다음과 같다.

1) Understanding the Requirements (12/07-12/08)

Project 안내 PDF와 pintOS Manual을 토대로 본 Phase에서 요구하는 Extensible Files & File Growth & Subdirectory를 비롯한 각 개발 사항의 이론적 개념을 이해한다. 이 중, 근래 강의 내용과 밀접한 관련을 지니는 FCB(File Control Block, a.k.a inode), Buffer Cache (Clock Algorithm) 등은 강의자료를 중점적으로 복습해 체득했다. 가장 핵심적이었던 부분은 On-Disk Structure 그 자체를 이해하는 것으로, File System이 전반적으로 어떻게 돌아가는지를 개괄적으로 이해할 수 있었다.

2) Extensible Files & File Growth & Buffer Cache (12/08-12/21)

기존 inode Structure에 Index Structure를 도입해 Direct, Indirect, Double Indirect Pointer를 마련하고, 그것을 토대로 pintOS File System의 External Fragmentation 문제를 해소하는 과정이다. 이론적 기반과 논리 구조는 어렵지 않았으나, 워낙 작성해야 할 코드 자체가 방대해 디버깅이 굉장히 까다로

웠다. 다 차례의 Troubleshooting이 필요했다.

한편, **Buffer Cache**의 구축도 이와 거의 동일 시점에 진행했으며, 그 구현 자체는 간단했다. 핵심은 Buffer Cache를 기준 inode Management에 병합하는 과정이었는데, 이 역시 다행히 I/O 부분만 신경 쓰면 되었기에 그렇게 고되진 않았다.

이렇게 두 Part의 구현은 시험 기간에 걸쳐서 진행되었으며, 따라서 약 2주 정도의 기간이 소요되었다.

3) Subdirectory (12/21-12/24)

시험이 끝난 후부터는 Subdirectory의 구현에 집중했다. **Subdirectory**의 근간이 되는 ‘inode 간의 Tree 구조 구축’ 그 자체의 구현은 상당히 간단했으나, 이를 **System Call**을 위시한 **User-Interface**로 완성하는 과정이 힘들었다. 가장 주요했던 부분은 아무래도 File System에서 Path Name들에 대한 String 처리와 Directory 관계 구축을 동시에 진행하는 부분이었으며, 그 과정에서 미묘한 에러가 자주 발생해 곤혹을 치렀다. 이는 다 차례의 Debugging과 Troubleshooting을 통해 완수할 수 있었다. 약 4일 가량 소요되었다.

전반적으로 본 Project Phase는 기반 이론 자체는 간단하지만 그것을 코드로 구현하는 것이 물리적으로 양이 많고 힘들었으며, 그 과정에서 오류가 발생했을 때 그 원인을 찾기가 어려운 점이 굉장히 까다로웠다. 모든 Test Case의 통과를 확인한 시점은 12월 24일 전후였으며, 이후 이를 정도 (12/24-12/25) 보고서 작성 및 코드 정리를 수행하였다.

B. 개발 방법

- Extensible File & File Growth

지금까지의 설명처럼, Extensible File & File Growth의 구현은 곧 **FCB**, 즉, **inode Structure**에 **Index** 기법을 적용해 다양한 **Indirect Pointer**를 두어 **inode**가 가리키는 **File**의 **Size**에 가변성을 주는 것이 핵심이었다. 그 말은 즉, 본 개발 항목은 **filesys/inode.h**가 관장한다.

가장 먼저 할 일은 기본적인 inode Format, 즉, '**inode_disk**' 구조체를 변형하는 것이다. 구조체에 Direct, Indirect Pointer들을 마련하고, 각 종 Metadata를 포함시키는 작업이 필요하다. 이때, On-Disk inode와 In-Memory inode를 꼭 구분하도록 하자. inode가 Disk에서 Memory로 옮겨졌을 땐 몇 가지 추가 정보(이를 테면 Reference Count 값)를 더 기억해야 File System 관리를 효과적으로 할 수 있다. 이는 '**inode**' 구조체에 '**inode_disk**' 구조체의 Field를 함께 두고, 거기에 추가 변수들을 둠으로써 구현할 수 있겠다.

이때, In-Memory inode에는 각 inode에 대해 Mutual Exclusion을 제공할 수 있는 Mutex Lock도 마련해두자. Directory에 대한 inode를 찾아가는 과정이나 inode를 확장하는 과정에서 동일 inode에 대한 Race가 있을 수 있기 때문이다.

이렇게 inode에 대한 구조체 정의가 완료되면 본격적으로 각 **inode** 관련 함수들을 수정한다. Disk 내 Sector Offset을 반환하는 'byte_to_sector' 함수부터 시작해서, 'inode_create', 'inode_open', 'inode_close' 등의 함수를 모두 수정한다. 각 함수의 기능에 따라 차이가 있지만, 이러한 수정은 모두 공통적으로 '**Pointer**의 종류에 따라 처리가 달라짐'이란 특징을 보인다. Direct Pointer에 대해선 곧 바로 그 Pointer가 가리키는 Block으로 이동해 기능을 수행하고, Indirect Pointer에 대해선 해당 Pointer가 가리키는 Block으로 이동 후, 해당 Block 내의 모든 가능 Pointer(128개)를 훑으며 연산을 제공한다. Double-Indirect의 경우엔 Indirect의 처리를 한 번 더 Nesting하면 된다. 모든 inode 관련 함수 수정은 이러한 궤를 공유한다.

inode가 가리키는 File에 대한 I/O를 수행하는 'inode_read_at', 'inode_write_at' 함수의 수정도 필요하다. 이땐 Offset 추출 작업 말곤 그다지 어려운 점은 없으나, 연산의 대상이 직접적으로 Disk를 향하지 않고 Buffer Cache를 거친다는 점을 기억하자. 자세한 내용은 후술한다.

또한, 본 파트에서 가장 중요한 'Extension 함수'도 정의해야 한다. 본인은 '**inode_extend**'라는 이름의 함수를 설계했다. 이 함수는 먼저 현재 다뤄지고 있는 inode의 Disk 내 Offset과 File Size를 따져 확장 필요성을 검사한다. 확장이 필요하다 판단되면, 곧 이어 Direct Pointer부터 쭉 돌며 공간을 할당하고, Disk Write를 수행한다. 이러한 'inode_extend' 함수가 **inode Creation 상황뿐만 아니라 inode I/O 상황에서도 호출되어 확장 필요 시 확장을 수행하면 된다.** 이렇게 하여 Extensible File의 구현이 완성되는 것이다.

- Subdirectory

Subdirectory 기능의 구현은 당연 **filesystem/directory.h**에서 관리한다. 본인은 **Directory 간의 계층 구조를 inode Structure에 포인터 링크를 둘으로써 구현할** 것이기에 가장 먼저 inode.h의 'inode_disk' 구조체와 'inode' 구조체를 변형한다. 이어, 동일 헤더 파일에, 특정 inode의 Parent inode를 반환하는 함수를 정의한다.

이렇게 기본 준비가 끝나면, directory.h로 이동해 Directory 관련 함수들을 상황에 맞게 수정한다. 크게 수정할 부분은 없으나, 주목해야 할 부분은 바로 'dir_add' 함수에서 새로운 Directory가 추가될 때 해당 Directory의 Corresponding inode의 Parent Link를 설정해주는 작업이다. 이를 통해 Directory 계층 구조가 만들어져 Subdirectory 기능의 제공이 가능해지기 때문이다.

그 밖에 'dir_remove' 함수에서 Directory 종류에 따른 처리를 분기하는 식의 수정 작업을 하자. 일부 함수, 일례로 'dir_readdir'과 같은 함수에선 동일 Directory inode에 대한 Race가 펼쳐질 수 있기에 앞서 언급했던 Mutex Lock을 이용해 Mutual Exclusion을 제공해야 함도 주목할 만하다.

이렇게 Directory 기반 함수들을 수정했으면, 이제 본격적으로 Directory 관련 Interface를 수정해야 한다. 무슨 말이냐 하면, **User에게 chdir, mkdir과 같은 Directory 관련 연산을 제공하는 User-Interface의 구축은 Directory 기반 함수 구성과 별개**라는 것이다. Interface가 기반 함수들을 이용해 기능을 제공하는 식인데, 이는 filesystem/filesys.h의 함수들을 수정하여 실현할 수 있다. 이때, 이들의 수정은 모두 다음과 같은 공통점을 보인다.

"Directory가 현재 Current Working Directory인지, 또는 그 CWD의 Parent Directory인지, 또는 아예 임의의 Directory인지'를 판단해 그에 따라 각기 다른 처리를 구사한다."

주로, 함수가 현재 인자로 받은 Directory가 CWD인 경우엔 특별한 수정을 가지 않고, CWD의 Parent이면 그 Directory의 Parent inode를 반환하며, 그 외 나머지 Case에 대해선 Directory Lookup을 수행하여 inode를 얻고, 거기에 기능을 제공하는, 그런 식으로 이뤄진다. 대표적으로 'filesystem_open' 함수의 수정이 이렇다.

한편, 기존 pintOS엔 아예 정의되어 있지 않은 '**filesys_chdir**'과 같은 함수를 정의하는 일도 필요하다. 이 함수는 당연 'chdir System Call'을 구현하는 함수이며, 그 구현 Flow는 앞서 말한 '공통점'을 그대로 가진다. Directory가 무엇이냐에 따라 Directory Change를 어떻게 할지 판단하면 된다. CWD인 경우엔 특별한 처리가 필요 없음이 자명하다.

이때, 이렇게 **Directory** 관련 **Interface**의 제공에 앞서 가장 먼저 선행되어야 할 일은 당연 **File Name String**에 대한 처리이다. File Name의 형식에 따라 String을 어떻게 Parsing하고, 거기서 Directory와 Name을 뽑아낼 것인지 고민하는 것이 중요하다. 이는 [filesys.h](#)에 '**path_get_dir**', '**path_get_file_name**'이란 이름의 함수를 정의해 구현하였다.

참고로, System의 모든 Thread는 자신의 Current Working Directory가 있고, chdir의 구현을 위해선 이를 기록하는 것이 필요하기에 threads/thread.h의 Thread Structure에 관련 멤버 변수 'dir'을 두는 것도 주목하자. 그리고, 상기 모든 구현 과정에서 'inode'가 가리키는 것이 Directory인지, 일반 File인지'를 알려주는 Flag를 inode Structure에 두고, 이를 '**inode_get_is_dir**'라는 식의 함수로 추출할 수 있게 환경을 만드는 것도 중요하다. 이는 'isdir System Call'의 구현에도 쓰일 것이다.

이밖에 'inumber System Call'과 같은 몇 가지 추가적인 System Call 등을 구현하면 Subdirectory 파트의 구현도 마무리된다.

- **Buffer Cache**

pintOS Manual에 따르면 Buffer Cache의 구현은 filesys Directory에 새로운 헤더 파일을 생성해 수행하라고 되어 있다. 따라서, '**cache.h**'라는 헤더 파일을 만든다. 그 내부엔 Buffer Cache의 사용자에게 제공할 Interface들을 정의한다.

이어, Buffer Cache를 구현한다. 가장 선행할 작업은 Buffer Cache의 Slot Format을 정의하는 것이다. 본인은 'cache_entry'라는 이름의 구조체를 만들어 이를 수행했으며, 그 내부엔 Freed, Accessed, Modified 여부를 나타내는 Flag 변수들을 포함해 몇 가지 데이터가 존재한다.

이를 토대로 간단한 배열 형태로 Buffer Cache를 만든다. 이어, Cache 초기화 함수, Cache Slot 접근 함수 등을 각각 'cache_init', 'cache_access_entry'라는 이름으로 정의한다. Cache에 새로운 Slot을 추가할 때 Cache를 순회했는데 더 이상 Available Slot이 발견되지 않는다면 Eviction을 수행하도록 설계하는 것이 중요하다.

이를 위해 우리는 Cache Eviction을 수행하는 함수도 정의한다. Clock Algorithm을 기반하며, 함수 이름은 '**cache_evict_entry**'로 하자. 단순한 Circular Traversal을 토대로 만들었으며, 이하 코드 설명 항목에서 간단히 확인해보자.

이 밖에 Cache Slot이 가리키는 Block에 데이터를 실제로 쓰거나 읽는 함수도 정의한다. 이러한 함수들의 동작 간에 Buffer Cache는 Shared Resource이므로 동기화가 보장되어야 함을 잊지 말자. 본인은 '**cache_lock**'이란 이름의 Mutex Lock을 도입해 이를 처리했다.

그리고, Buffer Cache Flush 역시 구현해야 함을 잊지 말자. Flush 자체는 간단하다. Buffer Cache를 Linear Traversal하면서 Dirty Slot만 Disk에 Update하면 된다. 이 Procedure가 어느 시점에 수행되느냐가 관건인데, 여태 설명했던 것처럼 이는 OS가 비정상 종료될 때 이뤄지며, 이는 filesystem/filesys.h의 '**filesystem_done**' 함수에 Flush를 설치함으로써 구현할 수 있다.

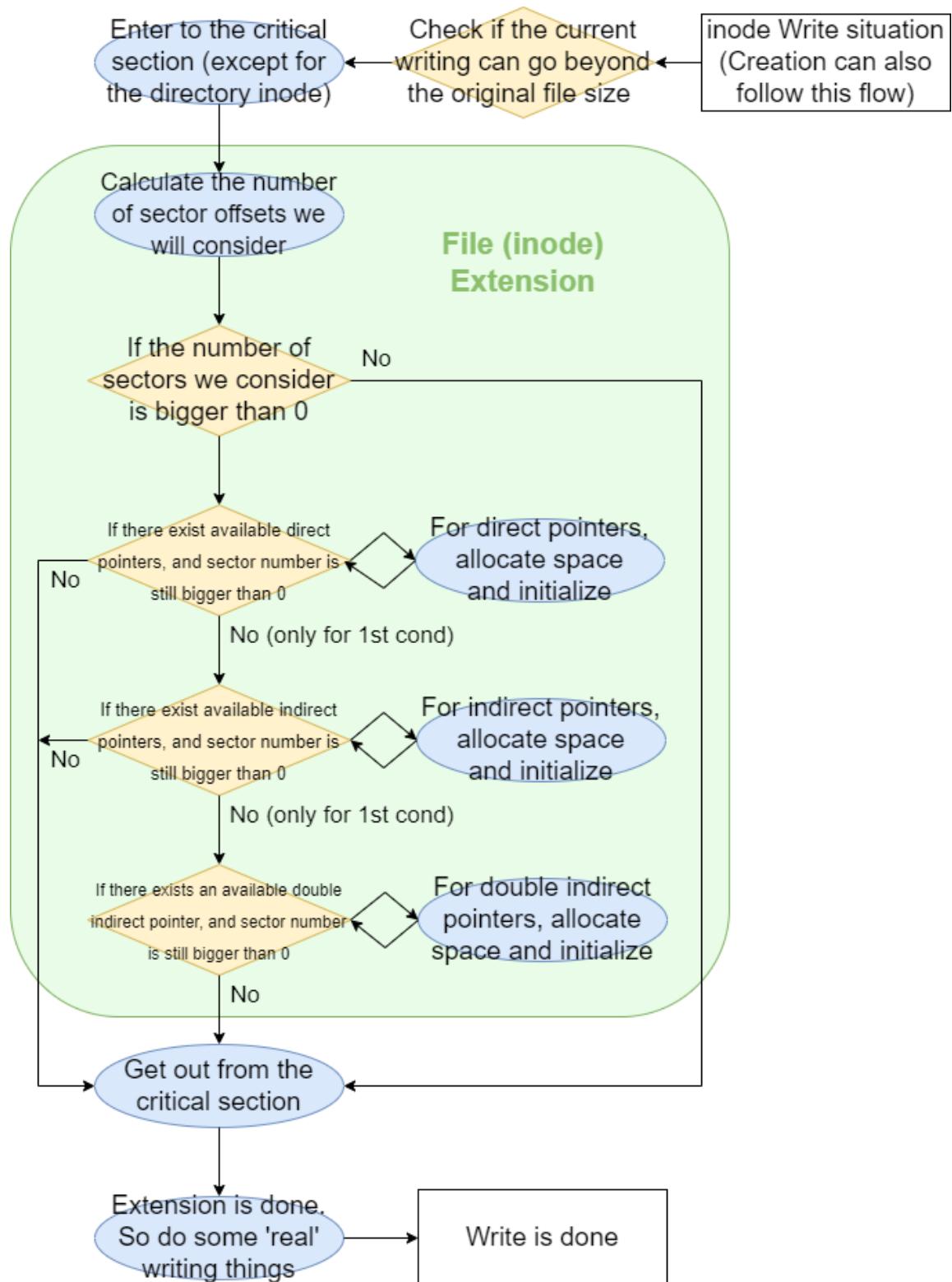
이렇게 해서 Buffer Cache를 가볍게 구현할 수 있다. 앞서 언급한 것처럼, Buffer Cache의 실제 사용은 결국 Disk I/O 상황이기 때문에 상기한 Buffer Cache 접근 함수를 inode.h의 Read / Write 함수에서 호출함으로써 사용할 수 있겠다. 그리고, 그러한 사용을 위해 가장 먼저 해야 할 일은 Buffer Cache를 File System에 설치하는 것이고 이는 filesystem/filesys.h의 `filesystem_init` 함수에서 Buffer Cache Initialization 함수를 호출함으로써 수행할 수 있겠다.

이어지는 다음 장에서부턴 본 Project Phase 5의 Flow Chart들을 소개한다.

4. 연구 결과

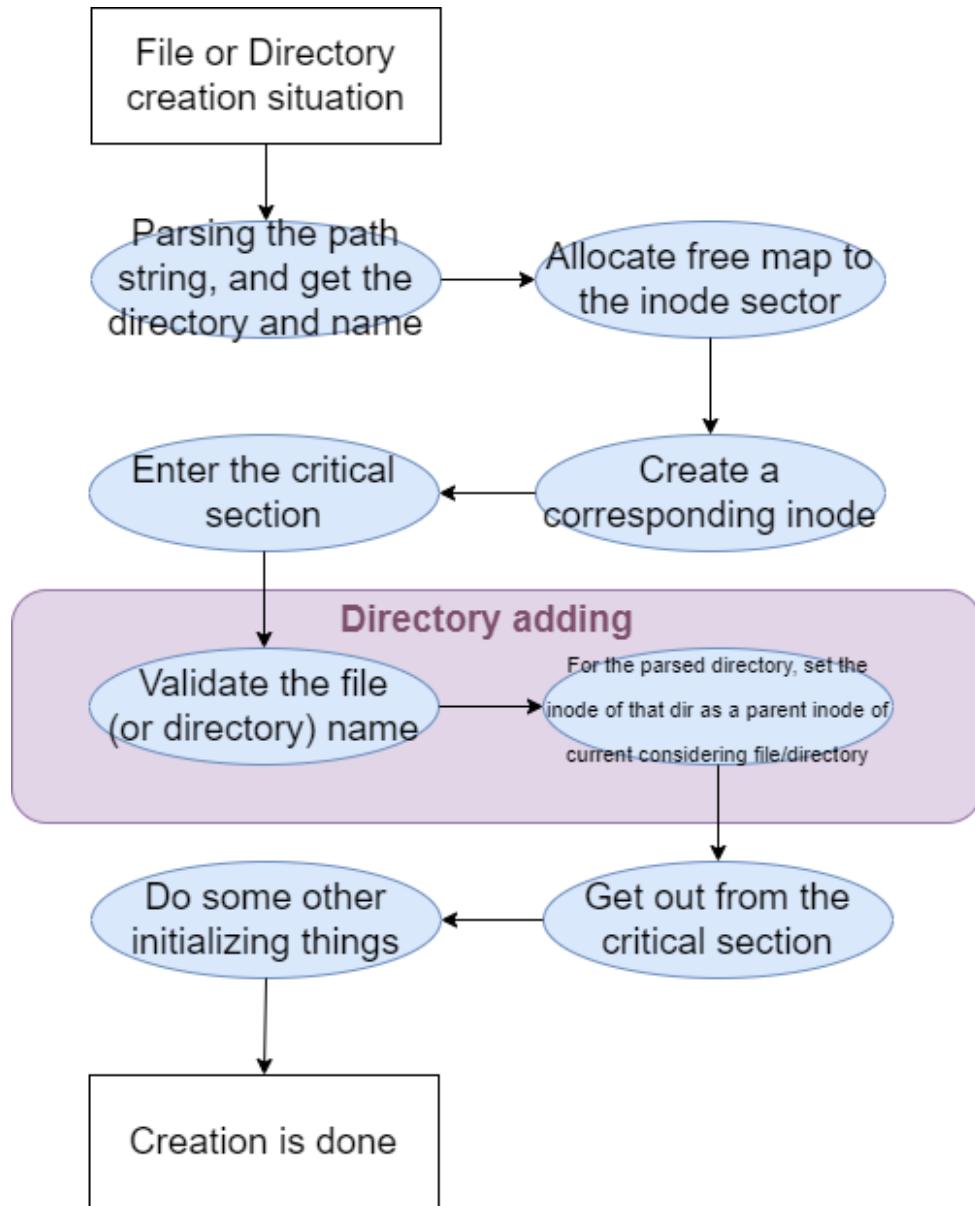
A. Flow Chart

- Extensible file & file growth



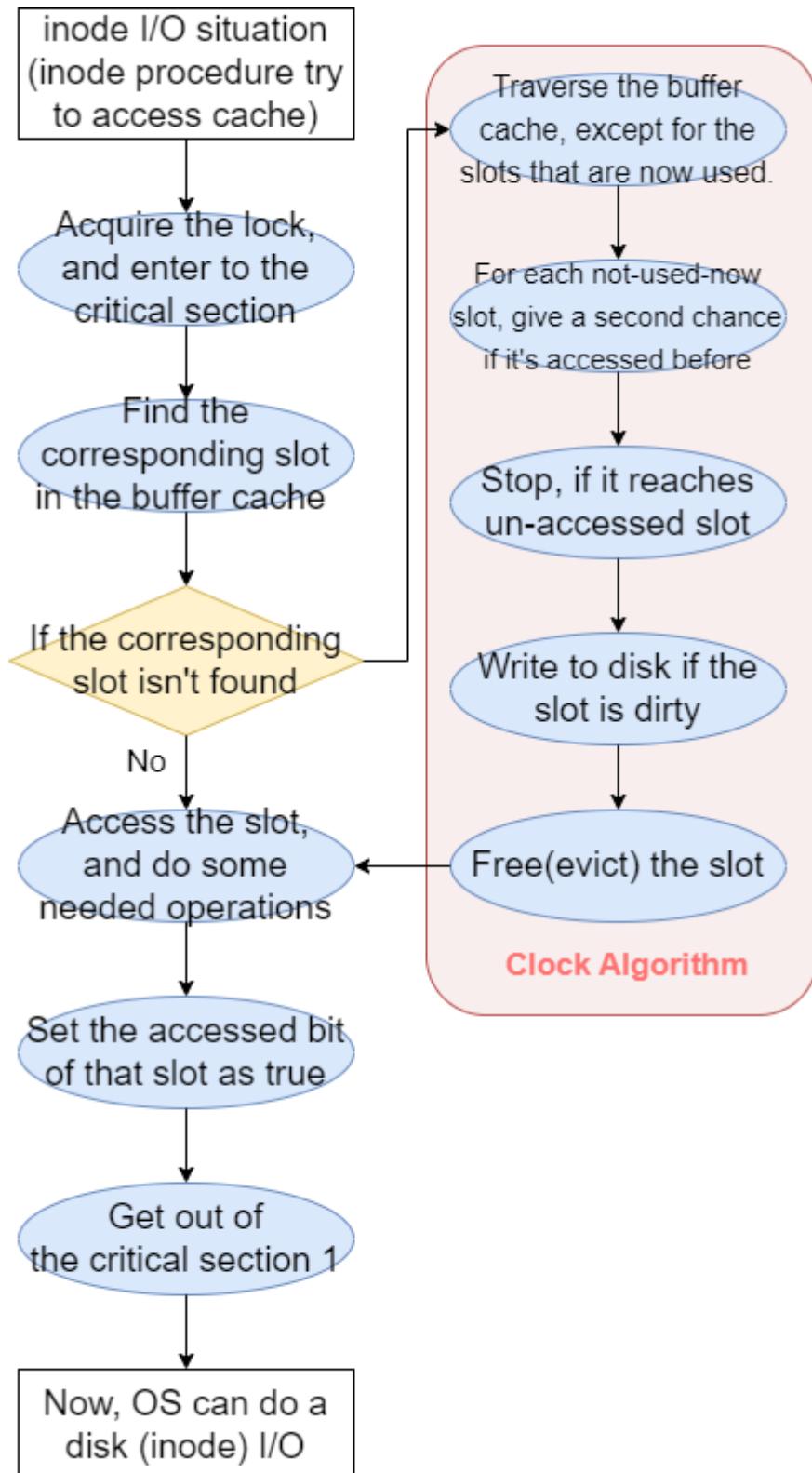
(inode와 관련한 모든 함수에 대한 Flow Chart 작성은 현실적으로 어렵다 판단하여 Extensible File & File Growth의 핵심인 'inode Write 시의 inode Extension' 상황만을 표현하였다.)

- Subdirectory



(역시나 마찬가지로 Directory와 관련한 모든 함수에 대해 Flow Chart를 그리긴 현실적으로 어려워 Subdirectory 관계, 'Directory 계층 구조(Tree)'를 구축하는 핵심 과정에 해당하는 부분(pintOS 함수 이름 기준 'filesystem_create')만 Flow Chart를 그렸다.)

- Buffer cache



(핵심 부분인 Eviction (Clock Algorithm) 부분만 Flow Chart로 묘사하였다.)

B. 제작 내용

지난 Project들과 마찬가지로 본 Phase에서도 본인의 제출 코드 내에 다양한 주석 문구로 코드 설명이 이뤄져 있다. 따라서, 본 보고서가 다루지 않은 부분에 대한 설명은 주석으로 대체한다. 주석을 통해서도 충분히 이해할 수 있다. 아래와 같이 열심히 달아놓았다.

```
/* The most notable feature of the new inode structure of
project 5 phase is that it can handle extensible files.

Before the fifth phase, the former linear file system of pintOS
suffered the existing external fragmentation, as written in
the manual PDF. And also it suffers impossible file growth problem.

Now, these problems have been solved by on-disk structure below.

+-----+
|on-disk | root-dir |           | root-dir |           |
|  inode |  on-disk | bit map |  entries |       free space
|        |    inode |           |           |           |
+-----+
~> FCB(File Control Block, a.k.a 'inode')s reside in free space,
   with related data sectors.
~> Each inode structure is sector-sized, that is 512B. It means
   there can be up to 128 indirect block pointers.
~> But you know, there're some additional data in inode-sector,
   so we need a compact design.
~> I choose to design with inode-sector of 16 block pointers
   consist of 5 direct pointer, 10 indirect pointers, and only
   one double indirect pointer.
~> With these pointers and 'indexed inodes' concept, problems
   of former pintOS will be resolved, since it allows files to
   grow whenever free space is available.

The details will be introduced in comments in 'inode.c' file.
*/
```

- Extensible file & file growth

우선 앞서 여러 차례 설명한 것처럼, File의 사이즈에 가변성을 주기 위해선 근본적으로 inode Structure에 대한 수정이 필요하다. On-Disk와 In-Memory를 구분해서 말이다.

또한, Index Structure가 도입된다고 했다. 본인은 상기 그림의 주석 및 앞선 항목들에서 설명한 것처럼 16개의 복합 Pointer로 이를 구축한다고 했다. 그 중 5개는 Direct, 10개는 Indirect, 1개는 Double-Indirect Pointer이다.

아래의 그림과 같이 inode Structure를 정의하자.

```

/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long.
   As mentioned, there are 16 block pointers with 5 directs,
   10 indirects, and only one double indirect. This structure
   is combined with 'index inode' concept, like below. */
struct inode_disk
{
    /* Basic Information. */
    off_t length;                      /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    bool is_dir;                        /* Is it an inode for a directory? */

    /* Pointers in inode-sector. */
    block_sector_t blocks[INODE_PTRS];  /* Total 16 pointers. */
    uint32_t direct_idx;               /* 5 direct pointers. */
    uint32_t indirect_idx;             /* 10 indirect pointers. */
    uint32_t d_indirect_idx;           /* 1 double indirect ptr. */

    /* Parent of current inode. */
    block_sector_t parent;              /* For indicating subdirectories. */

    /* Not-used-area. */
    uint32_t unused[SECTOR_PTRS - (INODE_PTRS + 7)];
};

```

이 중 'parent' Field는 Subdirectory 구축과 관련한 Pointer Link로, 아래 항목에 서 설명한다.

이렇게 On-Disk inode 구조체를 정의 후, 아래와 같이 In-Memory 구조도 설계 한다. Reference Count를 비롯해 OS에서 File System 관리 시 유용한 파일 Metadata들이 저장되고 있음을 주목하자.

```

/* In-memory inode. */
struct inode
{
    /* Information user in RAM should know. */
    struct list_elem elem;            /* Element in inode list. */
    block_sector_t sector;            /* Sector number of disk location. */
    int open_cnt;                     /* Number of openers. */
    bool removed;                     /* True if deleted, false otherwise. */
    int deny_write_cnt;               /* 0: writes ok, >0: deny writes. */
    off_t length;                     /* File size in bytes. */
    off_t read;                       /* How many bytes are read? */
    bool is_dir;                      /* Is it an inode for a directory? */

    /* Pointers in inode-sector. */
    block_sector_t blocks[INODE_PTRS]; /* Total 16 pointers. */
    uint32_t direct_idx;              /* 5 direct pointers. */
    uint32_t indirect_idx;             /* 10 indirect pointers. */
    uint32_t d_indirect_idx;           /* 1 double indirect ptr. */

    /* Parent of current inode. */
    block_sector_t parent;

    /* Mutex lock for accessing this inode. */
    struct lock inode_lock;
};

```

자, 이렇게 inode 구조체가 완성되었으면 먼저 'byte_to_sector' 함수를 아래와 같이 변형하자. 주어진 inode가 가리키는 File Size와 현재 고려하고 있는 File Position 간의 관계를 Index Structure를 기반으로 따져, 특정 File Position의 Data 가 inode가 Pointing하는 Block 중 어느 Block에 들어가 있는지를 찾아낼 수 있다.

```

static block_sector_t
byte_to_sector (const struct inode *inode, off_t length, off_t pos)
{
    ASSERT (inode != NULL);
    if (pos < length)
    {
        uint32_t idx; uint32_t blocks[SECTOR_PTRS];

        /* If the given position is inside the direct blocks,
           then simply return the corresponding direct slot. */
        if (pos < DIRECTS * BLOCK_SECTOR_SIZE)
            return inode->blocks[pos / BLOCK_SECTOR_SIZE];

        /* If the given position is inside the indirect blocks,
           then go through the chain with reading corresponding
           indirect blocks. */
        else if (pos <
                  (DIRECTS + INDIRECTS * SECTOR_PTRS) * BLOCK_SECTOR_SIZE)
        {
            /* Find the position we want to read. */
            pos -= (DIRECTS * BLOCK_SECTOR_SIZE);
            idx = (pos / (SECTOR_PTRS * BLOCK_SECTOR_SIZE)) + DIRECTS;

            /* Read. */
            block_read (fs_device, inode->blocks[idx], &blocks);

            /* Return the corresponding sector. */
            pos %= (SECTOR_PTRS * BLOCK_SECTOR_SIZE);
            return blocks[pos / BLOCK_SECTOR_SIZE];
        }

        /* If the given position is inside the double-indirect block,
           then same as the indirect block case, only except for that
           here's two levels. That is, we need read twice here. */
        else
        {
            /* First Level. */
            block_read(fs_device, inode->blocks[INODE_PTRS - 1], &blocks);

            /* Second Level. */
            pos -= ((DIRECTS + INDIRECTS * SECTOR_PTRS)
                    * BLOCK_SECTOR_SIZE);
            idx = pos / (SECTOR_PTRS * BLOCK_SECTOR_SIZE);
            block_read(fs_device, blocks[idx], &blocks);

            pos %= (SECTOR_PTRS * BLOCK_SECTOR_SIZE);
            return blocks[pos / BLOCK_SECTOR_SIZE];
        }
    }

    /* System should not reach here. */
    return INODE_ERROR;
}

```

이제 모든 준비는 마쳤다. 이제 inode 관련 기반 함수들을 Index Structure에 맞게 수정하기만 하면 된다. ‘inode_create’, ‘inode_open’과 같은 함수들의 Flow는 제출 코드에서 확인하자. 본인은 공간상의 이유로 본 보고서에서는 Extensible Files Concept의 핵심이 되는 부분만 축약해서 설명할 것이다.

아래는 특정 File에 대한 Write Operation 수행 시 자동으로 호출되는 inode Write 함수 ‘inode_write_at’ 함수이다.

```
/* Writes SIZE bytes from BUFFER into INODE, starting at OFFSET.
   Returns the number of bytes actually written, which may be
   less than SIZE if end of file is reached or an error occurs.
   (Normally a write at end of file would extend the inode, but
   growth is not yet implemented.) */
off_t
inode_write_at (struct inode *inode, const void *buffer_, off_t size,
                 off_t offset)
{
    const uint8_t *buffer = buffer_;
    off_t bytes_written = 0;

    if (inode->deny_write_cnt)
        return 0;

    /* Check if this writing can pass beyond EOF,
       if yes, extend the file. (file growth) */
    if (inode_length (inode) < (offset + size))
    {
        /* We should not give a mutual exclusion for dir-s. */
        if (!(inode_get_is_dir (inode)))
            lock_acquire (&(inode->inode_lock));

        /* File growth now has been made! */
        inode->length = inode_extend (inode, offset + size);

        if (!(inode_get_is_dir (inode)))
            lock_release (&(inode->inode_lock));
    }

    while (size > 0)
    {
        /* Sector to write, starting byte offset within sector. */
        block_sector_t sector_idx = byte_to_sector
            (inode, inode_length(inode), offset);
        int sector_ofs = offset % BLOCK_SECTOR_SIZE;

        /* Bytes left in inode, bytes left in sector, lesser of the two. */
        off_t inode_left = inode_length (inode) - offset;
        int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
        int min_left = inode_left < sector_left ? inode_left : sector_left;

        /* Number of bytes to actually write into this sector. */
        int chunk_size = size < min_left ? size : min_left;
        if (chunk_size <= 0)
            break;
```

```

/* In this pintOS, we use the buffer cache every time
we try to I/O with disk, just like below.
Calculate the buffer index, and then copy the memory.
Note that we should set the dirty flag of buffer slot. */
int buffer_idx = cache_access_entry (sector_idx, true);
cache_write_entry (buffer_idx, bytes_written, buffer,
sector_ofs, chunk_size);

/* Advance. */
size -= chunk_size;
offset += chunk_size;
bytes_written += chunk_size;
}

/* Record the read bytes. */
inode->read = inode_length(inode);
return bytes_written;
}

```

우리가 주목해야 하는 부분은 inode가 가리키는 File의 길이가 현재 Write를 할 범위를 커버할 수 있는지 확인하는 부분이다. 만약 그렇지 않다면, 'inode_extend'라는 함수를 호출하고 있다. 이 함수는 앞선 항목에서도 언급했듯 File Extension을 수행한다. 아래와 같이 생겼다.

```

/* Extend the file size pointed by the given inode.
That is, we should consider extension in all the
pointers in the FCB structure. */
static off_t
inode_extend (struct inode *inode, off_t length)
{
    static char zeros[BLOCK_SECTOR_SIZE];
    size_t sector_num;

    /* Is it need to be extended? */
    sector_num = (bytes_to_sectors (length)
                  - bytes_to_sectors (inode->length));
    if (sector_num == 0) return length;

    /* For the direct pointers, allocate free maps. */
    while (inode->direct_idx < DIRECTS)
    {
        if (sector_num == 0)
            break;

        free_map_allocate (1, &(inode->blocks[inode->direct_idx]));
        block_write (fs_device, inode->blocks[(inode->direct_idx)++],
                    zeros);
        sector_num--;
    }

    /* For the indirect pointers, allocate free maps. */
    while (inode->direct_idx < (DIRECTS + INDIRECTS))
    {

```

```

    if (sector_num == 0)
        break;

    block_sector_t blocks[SECTOR_PTRS];

    /* Allocate. */
    if (inode->indirect_idx == 0)
        free_map_allocate (1, &(inode->blocks[inode->direct_idx]));
    else block_read (fs_device, inode->blocks[inode->direct_idx],
                    &blocks);

    /* Expand. */
    while (inode->indirect_idx < SECTOR_PTRS)
    {
        if (sector_num == 0)
            break;

        free_map_allocate (1, &blocks[inode->indirect_idx]);
        block_write (fs_device, blocks[(inode->indirect_idx)++],
                     zeros);
        sector_num--;
    }

    /* Write expanded blocks back to the disk. */
    block_write (fs_device, inode->blocks[inode->direct_idx],
                 &blocks);

    /* Advance. */
    if (inode->indirect_idx == SECTOR_PTRS)
    {
        inode->indirect_idx = 0;
        inode->direct_idx++;
    }
}

/* For the d_indirect pointers, allocate free maps. */
if (inode->direct_idx == (INODE_PTRS - 1) && sector_num > 0)
{
    /* [0] is first level, [1] is second level. */
    block_sector_t blocks[2][SECTOR_PTRS];

    /* Allocate. (First Level) */
    if (inode->d_indirect_idx == 0 && inode->indirect_idx == 0)
        free_map_allocate (1, &inode->blocks[inode->direct_idx]);
    else block_read (fs_device, inode->blocks[inode->direct_idx],
                    &blocks[0]);

    /* Expand. (First Level) */
    while (inode->indirect_idx < SECTOR_PTRS)
    {
        if (sector_num == 0)
            break;

```

```

/* Allocate. (Second Level) */
if (inode->d_indirect_idx == 0)
    free_map_allocate (1, &blocks[0][inode->indirect_idx]);
else block_read (fs_device, blocks[0][inode->indirect_idx],
    &blocks[1]);

/* Expand. (Second Level) */
while (inode->d_indirect_idx < SECTOR_PTRS)
{
    if (sector_num == 0)
        break;

    free_map_allocate (1, &blocks[1][inode->d_indirect_idx]);
    block_write (fs_device,
        blocks[1][(inode->d_indirect_idx)++], zeros);
    sector_num--;
}

/* Write expanded blocks back to the disk. (Second Level) */
block_write (fs_device, blocks[0][inode->indirect_idx],
    &blocks[1]);

/* Advance. */
if (inode->d_indirect_idx == SECTOR_PTRS)
{
    inode->d_indirect_idx = 0;
    inode->indirect_idx++;
}
}

/* Write expanded blocks back to the disk. (First Level) */
block_write (fs_device, inode->blocks[inode->direct_idx],
    &blocks[0]);
}

return length;
}

```

코드가 굉장히 길다. 그러나, 사실 그 논리 구조는 굉장히 간단하다. 우선, 'inode_extend' 함수의 초반부를 보자. 앞서 확인한 'bytes_to_sectors' 함수를 통해 현재 Write를 할 전체 Size와 현재 inode가 가리키는 File의 File Size에 대해, Offset을 따져서 이들의 차이를 확인한다.

이 Sector Offset의 차이가 0보다 클 때에만 확장을 시도한다. 이는 굉장히 당연한 서술로, 만약 Sector Offset에 차이가 없다면, 그냥 현재 Sector에서 File의 Size만 원하는 Size로 바꾸어 Write를 수행하면 그만이다. 이미 커버되고 있기 때문이다.

Sector Offset이 0보다 큰 상황을 보자. 현재 다루고 있는 File의 inode에서 Direct Pointer부터 순차적으로 확인한다. 'direct_idx'와 같은 inode 구조체 Field는

현재 inode에서 바로 할당할 수 있는 Pointer Index를 의미하는데, 이 값이 DIRECTS Macro, Direct Pointer의 개수 이상이면 그 말은 즉 현재 inode의 Direct Pointer가 모두 사용되고 있다는 의미가 된다.

자, 이를 인지한 채 다시 코드를 보자. 그렇다. while문은 현재 사용되지 않고 있는 Direct Pointer를 찾고 있는 것이다. 그러한 Pointer가 있으면, 그 Pointer에 대해 Free Map Allocation 및 Block Writing을 통해 Block을 할당하는 것이다.

이어, Block(Sector)이 할당되면, 앞서 추출했던 Sector Offset을 Decrement한다. 그리고, 그 Sector Offset이 0인지 확인한다. 더 필요한지를 확인하는 것이다. 더 필요 없으면? 그렇다. 확장을 멈추면 된다.

논리가 굉장히 자명하다. 이어지는 Indirect Pointer, Double Indirect Pointer도 모두 마찬가지로 동작한다. 단지 차이는 Indirect의 경우, 해당 Pointer가 가리키는 Index Block을 모두 순회하며 일일이 Block 할당이 이뤄질 수 있다는 점이고, Double의 경우, 그러한 순회가 Nested 꼴로 이뤄진다는 것이다. 즉, Double의 경우엔 128 x 128개의 Block 할당이 가능한 것이다.

이렇게 해서 Expansion이 이뤄지는 것이다. 이렇게 확장된 inode를 토대로 Write Operation이 수행되게 된다. 이러한 'inode_extend'는 inode를 최초에 생성할 때에도 호출되어, 마치 Demand Paging처럼 inode Index Structure를 사용하고 있음을 보여준다.

```
/* Allocate a new space(in memory) for
   new inode, with growth is allowed. */
struct inode inode;
inode.length = 0; inode.direct_idx = 0;
inode.indirect_idx = 0; inode.d_indirect_idx = 0;

inode_extend (&inode, disk_inode->length);
disk_inode->direct_idx = inode.direct_idx;
disk_inode->indirect_idx = inode.indirect_idx;
disk_inode->d_indirect_idx = inode.d_indirect_idx;
memcpy (&(disk_inode->blocks), &(inode.blocks),
       INODE_PTRS * sizeof(block_sector_t));
```

한편, 앞선 'inode_extend' 코드에서 Index Structure를 훑는 Flow를 주목하자. 이는 그대로 inode Closing 상황에서 활용된다. 같은 흐름으로 가되, Allocation이 Deallocation으로 간다고 보면 된다. 대체로 inode 관련 함수 수정이 이렇게 진행된다.

이렇게 해서 inode에 관한 설명, File Extension에 대한 설명을 마친다. 나머지 관련 코드는 제출 코드 주석으로 설명을 대체한다.

- Subdirectory

자, 이번엔 Subdirectory가 코드 관점에서 어떻게 구현되는지 보자. 앞서 언급한 바와 같이, Subdirectory의 핵심은 결국 inode 간의 Link를 통한 Tree 구축이다. 즉, 이를 위해선 먼저 inode 구조체에서 parent 값을 추출하고 설정하는 루틴을 설계할 필요가 있다. 이를 다른 헤더에서 사용하게 하려는 목적이다.

```
/* Set the parent inode block. */
bool
inode_set_parent (block_sector_t parent, block_sector_t child)
{
    bool success = false;
    struct inode *inode = inode_open (child);
    if (!inode) return success;

    inode->parent = parent;
    inode_close (inode);

    return (success = true);
}

/* Returns the parent inode block. */
block_sector_t
inode_get_parent (const struct inode *inode)
{
    return inode->parent;
}
```

이를 토대로 Directory 연결 관계 구축이 어떻게 이뤄질까? 먼저 우리가 주목해야 할 부분은 directory.h가 아닌, filesystem.h이다. 왜냐? Directory의 생성은 누가 하는가? OS와 User이다. 특히 User의 mkdir 명령을 통해 생성된다.

따라서, 먼저 filesystem_create 함수를 볼 것이다. 그런데, 보기에 앞서, 먼저 사용자로부터 넘겨 받은 Path String을 Parsing하는 과정이 이뤄져야 한다. 예를 들어 사용자가 '/sogang/cse/junhyeok/pintos.c'라는 파일을 열고자 하고, 이렇게 Argument를 넘긴 경우, OS는 이를 '/'를 기준으로 Parsing하여, pintos.c라는 파일의 위치(Directory)가 'home->sogang->cse->junhyeok'임을 알고, 동시에 파일 이름은 'pintos.c'임을 알아야 하기 때문이다.

이는 아래와 같은 Parsing 함수들로 처리했다. filesystem.c에 static으로 정의한다.

```

/* Derives the directory location indicated by the given
   path string. It parses path string iteratively, and repeats
   open-close found directories, and thus the finally found
   directory will be the 'real directory location' of current file. */
static struct dir *
path_get_dir (const char *path)
{
    char path_name[PATH_MAX_LEN];
    char *cur, *temp, *prev;
    struct dir *dir; struct inode *inode;

    strlcpy (path_name, path, (strlen (path) + 1));

    /* Check the type of location representation. If the location
       start with '/' character, let's go through the path string
       with the start position of root directory. If not, it's the
       relative location representation. */
    if(path_name[0] == '/') || !(thread_current ()->dir)
        dir = dir_open_root ();
    else dir = dir_reopen (thread_current ()->dir);

    /* Parse the path string. */
    prev = strtok_r (path_name, "/", &temp);
    for (cur = strtok_r (NULL, "/", &temp);
         cur != NULL;
         prev = cur, cur = strtok_r (NULL, "/", &temp))
    {
        /* If it's the current working directory,
           then let's go to the next token. */
        if (!(strcmp (prev, ".") )) continue;

        /* If it's the parent directory of cwd,
           then get the inode of the parent dir. */
        else if (!(strcmp (prev, "..") )))
            dir_lookup_parent_inode (dir, &inode);

        /* If it's an arbitrary directory,
           then find the inode of that dir. */
        else dir_lookup (dir, prev, &inode);

        if (inode)
        {
            /* Close the recent searched directory,
               and then open the current directory. */
            if (inode_get_is_dir (inode))
            {
                dir_close (dir);
                dir = dir_open (inode);
            }
            else inode_close (inode);
        }
        else return NULL;
    }

    /* The final directory will be returned. */
    return dir;
}

/* Derives the file name from the given path string. */
static char *
path_get_file_name (const char *path)
{
    char path_name[PATH_MAX_LEN], *file_name;
    char *cur, *temp, *prev = "";

    /* Go to the final token. */
    strlcpy (path_name, path, (strlen (path) + 1));
    for (cur = strtok_r (path_name, "/", &temp);
         cur != NULL;
         prev = cur, cur = strtok_r (NULL, "/", &temp));

    /* The final token string will be a name of
       file indicated by the given path string. */
    file_name = malloc (strlen (prev) + 1);
    strlcpy (file_name, prev, (strlen (prev) + 1));

    return file_name;
}

```

이를 토대로 filesystem_create 함수를 아래와 같이 변형한다.

```

/* Creates a file named NAME with the given INITIAL_SIZE.
   Returns true if successful, false otherwise.
   Fails if a file named NAME already exists,
   or if internal memory allocation fails. */
bool
filesystem_create (const char *name, off_t initial_size, bool is_dir)
{
    block_sector_t inode_sector = 0;
    struct dir *dir = path_get_dir (name);
    char *file_name = path_get_file_name (name);
    bool success = (dir != NULL
                    && free_map_allocate(1, &inode_sector)
                    && inode_create(inode_sector, initial_size, is_dir)
                    && dir_add(dir, file_name, inode_sector));
    if (!success && inode_sector != 0)
        free_map_release (inode_sector, 1);
    dir_close (dir);
    free (file_name);

    return success;
}

```

그렇다. 사실은 변형된 것은 Parsing Method 호출 말고는 없다. 그런데, 우리는 여기서 'dir_add'라는 함수를 주목해야 한다. 해당 함수는 directory.h에 정의된 함수로 아래와 같이 설계된다.

```
/* Adds a file named NAME to DIR, which must not already contain a
   file by that name. The file's inode is in sector
   INODE_SECTOR.
   Returns true if successful, false on failure.
   Fails if NAME is invalid (i.e. too long) or a disk or memory
   error occurs. */
bool
dir_add (struct dir *dir, const char *name, block_sector_t inode_sector)
{
    struct dir_entry e;
    off_t ofs;
    bool success = false;

    ASSERT (dir != NULL);
    ASSERT (name != NULL);

    /* Give a mutual exclusion over look-up. */
    inode_lock_acquire (dir_get_inode (dir));

    /* Check NAME for validity. */
    if (*name == '\0' || strlen (name) > NAME_MAX)
        goto done;

    /* Check that NAME is not in use. */
    if (lookup (dir, name, NULL, NULL))
        goto done;

    /* Link the inode of the parent to the inode of child(currently
       considering file/directory). That is, this part is the core
       of implementing subdirectory concept in this project phase. */
    if (!inode_set_parent (inode_get_inumber (dir_get_inode (dir)),
                          inode_sector))
        goto done;
```

함수 후반부는 생략했다. 새롭게 추가된 부분은 바로 "if (!inode_set ..." 부분으로, 여기서 바로 parent 값 설정이 이뤄지고 있음을 주목하자.

그렇다. 사용자가 File / Directory Creation을 수행하면 위의 filesystem_create 함수가 호출되고, 그 함수는 인자로 넘겨 받은 Path String을 Parsing한다. 절대 경로, 상대 경로를 고려해 여러 Case를 따진 후, 최종적으로 File이 담긴 Directory와 File의 Name을 특정한다.

그리고, 이어, filesystem_create는 그 Directory와 Name을 토대로 위와 같이 inode를 새롭게 생성하고, 그 inode와 Parent의 inode를 연결하는 것이다. 바로 이렇게

해서 inode 간의 'Parent-Child' Tree 구조가 만들어지는 것이다. 그리고 이는 곧 Subdirectory 개념이 되는 것이다.

물론, 이론적인 부분과 밀접한 코드들은 이렇게 짧고 간단한데, 사실 각 종 Test Case를 통과하기 위해선 여러 가지 조정 과정이 필요하다. 보고서에 물리적인 이유로 그것을 다 실은 순 없다 (코드 주석에 잘 설명되어 있음). 하지만, 그 중에서도 주요했던 부분은 몇 가지 소개하겠다. 아래를 보자.

```
/* Change the current working directory.  
 Returns true if successful, false on failure.  
 Fails if the given directory is not proper. */  
bool  
filesys_chdir (const char *name)  
{  
    struct dir *dir = path_get_dir (name);  
    char *file_name = path_get_file_name (name);  
    struct inode *inode = NULL;  
  
    if (dir != NULL)  
    {  
        /* If the given directory is cwd or root directory,  
         then you don't have to change anything. */  
        if ((strlen (file_name) == 0 && dir_is_root (dir)) ||  
            !(strcmp (file_name, ".") ))  
        {  
            free (file_name);  
            return true;  
        }  
  
        /* If the given directory is a parent of cwd,  
         then get the inode of that directory. */  
        else if (!strcmp (file_name, ".."))  
            dir_lookup_parent_inode (dir, &inode);  
  
        /* If the given directory is an arbitrary dir,  
         then find the inode of that directory. */  
        else dir_lookup (dir, file_name, &inode);  
    }  
    else goto error;  
  
    /* Now, close the former directory, and  
     open the new directory. */  
    dir_close (dir);  
    dir_close (thread_current ()->dir);  
    if ((dir = dir_open (inode)) == NULL)  
        goto error;  
    free (file_name);  
  
    /* Set the current working directory. */  
    thread_current ()->dir = dir;  
  
    return true;  
  
error:  
    /* Error occur. */  
    free(file_name);  
    return false;  
}
```

chdir System Call을 수행하는 함수이다. 우리는 본 Phase에서 chdir, mkdir을 비롯한 Directory 관련 System Call들, 그리고 일부 기존 File 관련 System Call들을 수정해야 한다. 그 중 이 chdir은 주목할만하다. 세 개의 Case로 상황을 분기해 처리하고 있기 때문이다. 이 파트의 함수 설계가 대부분 이러한 Flow로 이루어졌다.

자세히 봐보자. Caller로부터 넘겨 받은 File Name이 Root Directory 또는 Current Working Directory를 가리키면 아무런 Directory 이동을 하지 않는다.

반면, File Name이 Parent Directory를 가리킨다면, 해당 Directory의 inode를 추출한다. 마찬가지로, File Name이 위 두 상황에 해당하지 않는다면 전체 Directory에서 그 Directory를 찾아 해당 inode를 추출한다.

그렇게 추출한 inode를 가지고 새롭게 Directory를 Open한다. 이어, 기존의 Directory는 닫아버린다. 그리고, 현재 Thread의 CWD를 본 Directory로 지정하고 있다. 이렇게 chdir이 동작하는 것

이다. 이번엔 아래의 함수를 보자.

```
/* Create a directory.  
This function works just like the 'filesys_create', but only  
except for the creation part. 'filesys_create' creates a file  
with inode creation, but here, it uses directly a directory  
creation. */  
bool  
filesys_mkdir (const char *name)  
{  
    return filesystem_create(name, 16, true);  
}
```

그렇다. mkdir System Call을 구현한 함수이다. 앞서 언급한 것처럼 기본 File Creation 함수로 가볍게 수행할 수 있음을 주목하자. 이때, 한 가지 주의할 점은, File Creation 시 초기 사이즈를 위와 같이 어느 정도 크기로 정해 할당해야 한다는 점이다. 왜냐? Directory는 기본적으로 File System 전체 메모리 공간에서 감당 가능한 File들은 모조리 가리킬 수 있어야 하기 때문이다. 이런 말을 왜 할까? 그것은 바로, 본 Phase의 'dir-vine' Test Case가 바로 이러한 상황을 콕 집어 점검하기 때문이다.

이렇게 해서 Subdirectory 부분에 대한 코드 설명을 마친다. 역시나 앞선 항목과 마찬가지로 본 보고서에서 다루지 못한 부분의 설명(특히나 System Call 호출 부분들)은 제출 코드 주석으로 대체하겠다.

- **Buffer cache**

Buffer Cache는 File System에 설치된 'I/O 속도 향상을 위한 Cache'이다. 이는 기존 pintOS에는 없는 시스템으로, 새롭게 추가해야 한다. 따라서, 아래와 같이 cache.h라는 파일을 filesystem Directory에 생성한다. 그리고 'Makefile.build'도 수정해야 함을 기억하자.

아래는 cache.h 파일이다.

```

#ifndef FILESYS_CACHE_H
#define FILESYS_CACHE_H

#include "devices/block.h"
#include "threads/synch.h"
#include "fsys/off_t.h"

/* This header provides a buffer cache for enhancing an
   efficiency of disk I/O in the pintOS file system.

   The buffer cache here is implemented via 'Clock (Second Chance)
   Algorithm', which iterates the linear buffer in circular way,
   with finding the un-accessed sector-sized slot when the cache
   is full.

   When the system operates disk I/O, every request must be operated
   via 'inode structure' defined in 'fsys/inode.h', that is, inode.h
   will use this header file to give the system the efficient performance. */

/* Interfaces of buffer cache. */
void cache_init (void);
int cache_access_entry (block_sector_t disk_sector, bool is_dirty);
void cache_read_entry (int, off_t, uint8_t *, int, int);
void cache_write_entry (int, off_t, uint8_t *, int, int);
void cache_flush (void);

#endif

```

Buffer Cache에서 제공하는 Interface들을 확인할 수 있다.

이 중 우리는 'cache_access_entry'라는 Buffer Cache Slot 접근 함수를 주목한다.
먼저, Buffer Cache의 구조를 잠시 확인하자.

```

/* Format of each entry in the buffer cache. */
struct cache_entry
{
    /* Data part of each slot. */
    uint8_t block[BLOCK_SECTOR_SIZE]; /* The associated sector. */
    block_sector_t disk_sector;        /* The associated sector #. */

    /* Metadata of each slot. */
    int open_cnt;                      /* Reference count of slot. */
    bool is_free;                      /* Is this slot freed? */
    bool is_accessed;                 /* Is this slot accessed? */
    bool is_dirty;                     /* Is this slot modified? */
};

/* Maximum size of buffer cache. */
#define CACHE_SIZE 64

/* Buffer cache. */
struct cache_entry cache[CACHE_SIZE];

/* For a mutual exclusion of buffer cache. */
struct lock cache_lock;

```

Slot은 하나의 Disk Sector와 맵핑되고, 거기에 추가로 Reference Count와 Freed, Accessed, Modified Flag를 가짐을 주목하자. 이 Flag들은 모두 Clock Algorithm의

구현에 사용된다. 이러한 Slot Format을 토대로 크기 64의 Buffer Cache가 만들어진다.

잠시 멈추어, 앞서 확인했던 'inode_write_at' 함수를 다시 봄자. 해당 코드의 Write 부분은 아래와 같이 이뤄졌었다.

```
/* Sector to write, starting byte offset within sector. */
block_sector_t sector_idx = byte_to_sector
    (inode, inode_length(inode), offset);
int sector_ofs = offset % BLOCK_SECTOR_SIZE;

/* Bytes left in inode, bytes left in sector, lesser of the two. */
off_t inode_left = inode_length (inode) - offset;
int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
int min_left = inode_left < sector_left ? inode_left : sector_left;

/* Number of bytes to actually write into this sector. */
int chunk_size = size < min_left ? size : min_left;
if (chunk_size <= 0)
    break;

/* In this pintOS, we use the buffer cache every time
   we try to I/O with disk, just like below.
   Calculate the buffer index, and then copy the memory.
   Note that we should set the dirty flag of buffer slot. */
int buffer_idx = cache_access_entry (sector_idx, true);
cache_write_entry (buffer_idx, bytes_written, buffer,
    sector_ofs, chunk_size);

/* Advance. */
size -= chunk_size;
offset += chunk_size;
bytes_written += chunk_size;
```

그렇다. Write가 Disk에 직접 바로 이뤄지는 것이 아니라 Buffer Cache의 Slot에 대해 이뤄지고 있는 것이다. Buffer Cache의 Slot에 Access하고, 거기서 Write하고 있는 것이다.

그러면, 이 Access는 어떻게 이뤄지는지 봄자. 아래와 같다.

```
/* Access the slot of buffer cache, indicated by the given sector #.
   That is, only inode operations call this interface.
   Note that before reading or writing the slot, inode structure must
   access the slot first by calling this function. */
int
cache_access_entry (block_sector_t disk_sector, bool is_dirty)
{
    int idx = -1, i;

    lock_acquire (&cache_lock);

    /* First, find the corresponding slot in the buffer cache. */
    for (i = 0; i < CACHE_SIZE; i++)
```

```

    {
        if (cache[i].disk_sector == disk_sector &&
            cache[i].is_free == false)
        {
            idx = i;
            break;
        }
    }

/* If there's no freed slot, that is, the cache is
   full, then let's do an eviction! */
if (idx == -1)
    idx = cache_evict_entry (disk_sector, is_dirty);
else
{
    cache[idx].open_cnt++;
    cache[idx].is_accessed = true;
    cache[idx].is_dirty |= is_dirty;
}

lock_release (&cache_lock);

return idx;
}

```

Buffer Cache를 순회하여 Access하고자 하는 Slot을 찾는다. Slot이 발견되면 Reference Count를 Increment하고, Access Bit를 Set한 다음, I/O를 할 준비를 한다. 그런데 만약 Slot이 발견되지 않는다면? 그렇다. Buffer Cache에서 Eviction이 수행된다. 아래와 같이 말이다.

```

/* Clock (Second Chance) Algorithm based eviction routine. */
static int
cache_evict_entry (block_sector_t disk_sector, bool is_dirty)
{
    int idx = -1, i; /* i is iterator. */

    /* First, check if there're any freed slot
       in the buffer cache. */
    for (i = 0; i < CACHE_SIZE; i++)
    {
        if (cache[i].is_free == true)
        {
            cache[i].is_free = false;
            idx = i;
            break;
        }
    }

    i = 0;
    /* If the cache is full, let's do an eviction. */
    if (idx == -1)
    {
        while (1)
        {

```

```

/* If the slot is currently used,
we should not touch it. */
if (cache[i].open_cnt > 0)
    continue;

/* Give a second chance to slot, if it's accessed. */
if (cache[i].is_accessed == true)
    cache[i].is_accessed = false;

/* If the slot isn't accessed before, then evict it. */
else
{
    /* If the slot is dirty, then write behind to disk. */
    if (cache[i].is_dirty == true)
        block_write (fs_device, cache[i].disk_sector,
                     &cache[i].block);

    /* Free the slot. */
    cache[i].open_cnt = 0;
    cache[i].is_free = true;
    cache[i].is_accessed = false;
    cache[i].is_dirty = false;
    idx = i;
    break;
}

/* Advance in a circular way. */
i = (i + 1) % CACHE_SIZE;
}
}

/* Replace each field with new one. */
cache[idx].disk_sector = disk_sector;
cache[idx].open_cnt++;
cache[idx].is_free = false;
cache[idx].is_accessed = true;
cache[idx].is_dirty = is_dirty;
block_read (fs_device, cache[idx].disk_sector,
            &cache[idx].block);

return idx;
}

```

먼저 Free Slot이 있는지를 점검한다. 만약 Free Slot이 남아 있다면 그대로 반환하면 되기 때문이다. 만약 그러한 Slot이 없다면? 바로 Clock Algorithm이 수행된다. Buffer Cache를 원형으로 순회하며 Reference Count가 0보다 큰, 즉, 현재 사용되고 있는 Block들은 제외하고, Accessed Block에 대해 Second Chance를 준다. 그러다 처음으로 Un-Accessed Block을 발견하면, 그대로 멈추어 해당 Slot을 Free한 후 반환하는 것이다. 이렇게 Buffer Cache의 Eviction이 이뤄진다.

Buffer Cache가 Eviction 등을 통해 여유 Slot을 제공하면, Write Operation은 그대로 쭉 진행된다.

이러한 Buffering은 Disk I/O 시의 비효율성을 낮추고, 성능 향상을 이룩하는 방법 중 하나로 알려진다.

이때, 우리는 Flush 상황도 만들어주어야 한다. 우선, 아래처럼 Flush Routine을 간단하게 만들어준다. Buffer Cache를 Linear하게 돌면서 Dirty Slot에 대해서만 Disk Update를 수행한다.

```
/* Total write-behind (flushing) routine of buffer cache.  
   That is, this function will be called when there's an  
   abnormal termination of pintOS, or unmounting of file system. */  
void  
cache_flush (void)  
{  
    int i;  
  
    lock_acquire (&cache_lock);  
  
    /* Iterate all the slots in the buffer cache, with  
       updating dirty blocks into the disk. */  
    for (i = 0; i < CACHE_SIZE; i++)  
    {  
        if (cache[i].is_dirty == true)  
        {  
            block_write (fs_device, cache[i].disk_sector,  
                        &cache[i].block);  
            cache[i].is_dirty = false;  
        }  
  
        cache[i].open_cnt = 0;  
        cache[i].is_free = true;  
        cache[i].is_accessed = false;  
        cache[i].is_dirty = false;  
    }  
  
    lock_release (&cache_lock);  
}
```

이러한 'cache_flush' 함수는 언제, 어디서 호출되어야 할까? 이미 앞서 한 차례 언급한 바 있듯, 이는 filesystem.c의 filesystem_done 함수에서 호출되어야 한다. 해당 함수는 File System Unmounting이나 OS Crash와 같은, (주로) 비정상 상황 시 File System이 종료되어야 할 때 호출되는 함수인데, 그러한 종료 시 항상 Buffer Cache를 Disk에 Flush하도록 만들어 주는 것이다.

```

/* Shuts down the file system module, writing any unwritten data
   to disk. */
void
filesys_done (void)
{
    /* Flush the buffer cache, in case
       of an abnormal situation. */
    cache_flush ();

    free_map_close ();
}

```

이를 통해 우리는 pintOS File System에 Persistency를 부여할 수 있다.

이렇게 해서 본 Project Phase 5에 대한 코드 설명을 마친다. 계속 언급하는 것처럼, 본 보고서에 공간 상의 이유로 소개하지 않는 코드에 대한 설명은 제출 코드의 주석으로 대체한다.

C. 시험 및 평가 내용

- **Src/filesys make grade 수행결과를 캡처 하여 첨부**

위와 같이 개발된 pintOS는 Phase 5에서 요구하는 모든 Test Case를 통과한다.
아래 좌측은 make check 결과의 일부이다. 그리고 우측은 make grade 결과이다.

```

pass tests/filesys/extended/dir-empty-name-persistence
pass tests/filesys/extended/dir-mk-tree-persistence
pass tests/filesys/extended/dir-mkdir-persistence
pass tests/filesys/extended/dir-open-persistence
pass tests/filesys/extended/dir-over-file-persistence
pass tests/filesys/extended/dir-rm-cwd-persistence
pass tests/filesys/extended/dir-rm-parent-persistence
pass tests/filesys/extended/dir-rm-root-persistence
pass tests/filesys/extended/dir-rmdir-persistence
pass tests/filesys/extended/dir-under-file-persistence
pass tests/filesys/extended/dir-vine-persistence
pass tests/filesys/extended/grow-create-persistence
pass tests/filesys/extended/grow-dir-lg-persistence
pass tests/filesys/extended/grow-file-size-persistence
pass tests/filesys/extended/grow-root-lg-persistence
pass tests/filesys/extended/grow-root-sm-persistence
pass tests/filesys/extended/grow-seq-lg-persistence
pass tests/filesys/extended/grow-seq-sm-persistence
pass tests/filesys/extended/grow-sparse-persistence
pass tests/filesys/extended/grow-tell-persistence
pass tests/filesys/extended/grow-two-files-persistence
pass tests/filesys/extended/syn-rw-persistence
All 125 tests passed.

```

TOTAL TESTING SCORE: 100.0%																																												
ALL TESTED PASSED -- PERFECT SCORE																																												

SUMMARY BY TEST SET																																												
<table border="1"> <thead> <tr> <th>Test Set</th><th>Pts</th><th>Max</th><th>% Ttl</th><th>% Max</th></tr> </thead> <tbody> <tr> <td>tests/filesys/extended/Rubric.functionality</td><td>34/ 34</td><td>30.0%</td><td>30.0%</td><td></td></tr> <tr> <td>tests/filesys/extended/Rubric.robustness</td><td>10/ 10</td><td>15.0%</td><td>15.0%</td><td></td></tr> <tr> <td>tests/filesys/extended/Rubric.persistence</td><td>23/ 23</td><td>20.0%</td><td>20.0%</td><td></td></tr> <tr> <td>tests/filesys/base/Rubric</td><td>30/ 30</td><td>20.0%</td><td>20.0%</td><td></td></tr> <tr> <td>tests/userprog/Rubric.functionality</td><td>108/108</td><td>10.0%</td><td>10.0%</td><td></td></tr> <tr> <td>tests/userprog/Rubric.robustness</td><td>88/ 88</td><td>5.0%</td><td>5.0%</td><td></td></tr> <tr> <td>Total</td><td></td><td></td><td>100.0%</td><td>100.0%</td></tr> </tbody> </table>					Test Set	Pts	Max	% Ttl	% Max	tests/filesys/extended/Rubric.functionality	34/ 34	30.0%	30.0%		tests/filesys/extended/Rubric.robustness	10/ 10	15.0%	15.0%		tests/filesys/extended/Rubric.persistence	23/ 23	20.0%	20.0%		tests/filesys/base/Rubric	30/ 30	20.0%	20.0%		tests/userprog/Rubric.functionality	108/108	10.0%	10.0%		tests/userprog/Rubric.robustness	88/ 88	5.0%	5.0%		Total			100.0%	100.0%
Test Set	Pts	Max	% Ttl	% Max																																								
tests/filesys/extended/Rubric.functionality	34/ 34	30.0%	30.0%																																									
tests/filesys/extended/Rubric.robustness	10/ 10	15.0%	15.0%																																									
tests/filesys/extended/Rubric.persistence	23/ 23	20.0%	20.0%																																									
tests/filesys/base/Rubric	30/ 30	20.0%	20.0%																																									
tests/userprog/Rubric.functionality	108/108	10.0%	10.0%																																									
tests/userprog/Rubric.robustness	88/ 88	5.0%	5.0%																																									
Total			100.0%	100.0%																																								

모든 Test Case를 문제 없이 통과하고 있다.

이렇게 해서 이번 가을 학기 내내 진행되었던 pintOS Project를 마친다. 그간 어렵게만 느껴졌던 운영체제 개념을 실제 코드 단위에서 어떻게 실현하는지를 몸소 느낄 수 있어 굉장히 의미 있는 시간이었다 생각한다. 그 과정은 상당히 어렵고 힘들었지만, 결국 열심히 하여 5개의 과제를 모두 잘 마무리하게 되어 개인적으로 많이 뿌듯하다. 이번 경험을 통해 체득한 실력, 이해를 토대로 앞으로도 열심히 공부하겠다.

지난 한 학기 동안 열심히 학생들을 지도해주신 교수님과, 바쁜 와중에도 학생들을 잘 돌봐주신 조교님들께 감사를 표하며 보고서를 마치겠습니다. 감사합니다. 좋은 연말, 그리고 새해가 되시길 바랍니다.