**Pintos Project 5: Filesystem**

담당 교수 : 박성용

조 / 조원 : 이선호

개발 기간 : 2022.12.09 ~ 2022.12.26

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

기존 PintOS의 파일 시스템에서 한 번 파일을 생성한 이후에 파일 크기를 늘릴 수 있도록 하며, 이를 위해 파일의 block이 비연속적인 공간 차지를 할 수 있도록 하여 external fragmentation의 문제를 해결한다. 또한 디렉토리 안에 디렉토리가 존재할 수 있는 서브 디렉토리 구조를 구현하고, 메모리에서 좀 더 빠르게 파일을 읽을 수 있도록 buffer cache를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Extensible file & file growth

기존 PintOS에서는 파일이 한 block 내에 저장되어서 한 번 생성된 이후로 파일 크기를 변경할 수 없지만, 더 큰 용량의 내용을 파일에 추가해야 할 때 제한된 파일 크기로 인해 제약이 생긴다. 그래서 필요에 따라 새로운 block을 더 추가하여 파일의 크기를 늘릴 수 있는 file growth 기능을 구현하여 용량에 제한 없이 데이터를 쓸 수 있도록 한다. 또한 비연속적인 block 단위로 파일을 관리함으로써 external fragmentation 문제를 해결할 수 있다.

1. Subdirectory

기존 PintOS에서는 root 디렉토리 하나에만 파일이 저장되었는데, 서브 디렉토리를 얼마든지 추가할 수 있도록 하여 다층적인 파일 디렉토리 구조를 지니도록 한다.

1. Buffer cache

디스크에 저장되어 있는 파일로부터 직접 데이터를 읽거나 쓰는 것은 overhead가 많이 든다. 그래서 필요에 따라 디스크에서 읽기와 쓰기 속도가 빠른 메모리로 적재해 와서 파일과 관련한 작업을 수행할 수 있는 buffer cache를 구현한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Extensible file & file growth
  + Index structure와 management에 대해서 기술

파일이 저장되어 있는 block을 단순히 contiguous allocation으로 시작 위치와 끝 위치만을 저장하는 방식을 사용하면 external fragmentation을 발생시킬 가능성이 높고 파일 크기 확장도 어려워진다. 이를 보완하는 방법이 바로 index structure인데, 각 파일마다 하나의 index block을 지니고 그 index block에서 파일과 관련된 block을 가리키는 방식이다. 이번 프로젝트에서 구현한 방식은 inode\_disk라는 구조가 index block과 유사하다.

Index structure에서는 direct block, single indirect block, double indirect block 이 세 가지 종류의 block을 사용하여 파일의 block을 관리한다. 파일을 생성할 때 파일이 필요로 하는 sector의 범위를 계산하여 필요에 따라 direct block, single indirect block, double indirect block으로 할당을 확장해 간다. 파일을 삭제할 때는 위의 세 가지 종류의 block이 가리키는 모든 file block에 접근하여 해제한다.

* Subdirectory
  + Directory entry 관리 방법

File 구조체 대신 Director entry를 사용하여 새로운 디렉토리를 생성할 때 기존의 root directory부터 차례로 찾아갈 수 있도록 가리켜서 부모와 자식 간의 관계를 갖게 한다. 또한 디렉토리 주소가 상대 경로인지 절대 경로인지를 파악 가능하게 하며, 파일 생성 시 해당 파일이 디렉토리인지 아닌지를 구분할 수 있는 is\_dir 멤버 변수를 사용한다. 그리고 현재 thread가 사용하고 있는 디렉토리가 어떤 것인지를 cur\_dir라는 변수를 활용하여 확인 가능하도록 한다.

* Buffer cache
  + Buffer cache eviction 방식

디스크로부터 메모리로 파일의 block을 가져와야 할 때 빈 자리가 없을 경우 기존의 buffer cache entry에서 어떠한 entry를 제외시켜야 하는 정책으로 크게 두 가지를 구현했다. 첫 번째는 clock algorithm으로 buffer cache entry를 circular하게 탐색하면서 valid bit과 reference bit이 모두 1이면 reference bit을 0으로 바꾸고, 그렇지 않은 경우 victim entry로 선택될 수 있도록 했다. 효율성은 좋지만 모두 다 valid bit과 reference bit이 1인 경우 cycle을 다시 돌아야 하는 overhead가 존재해서 최종적으로는 두 번째 방식을 택했다.

두 번째 방식은 전체 buffer entry에서 가장 늦게 access된 entry를 victim으로 택하는 LRU algorithm이다. 이를 위해 매번 read하거나 write할 때 access time 변수에 tick을 기록했으며, victim을 탐색할 때 가장 tick이 작은 entry를 택하도록 했다. 비록 매번 victim을 고를 때 전체 buffer cache entry를 탐색해야 하는 단점이 있지만, 평균적인 탐색 속도가 나쁘지 않고 실제 동작 시 이러한 overhead를 고려할 정도로 큰 문제가 되지 않는다고 판단하여 두 번째 방식을 최종적으로 택했다.

* + Buffer cache flush 방식

Buffer cache entry에 write할 때 dirty bit을 1로 설정하는데, buffer에 있는 파일 내용을 디스크에 써야할 때 이전에 dirty bit이 1로 된 buffer cache entry를 찾아서 모두 디스크로 flush한다. Flush 이후에는 flush한 buffer cache entry를 valid bit, reference bit, dirty bit을 모두 0으로 초기화한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

2022. 12. 09 ~ 2022. 12. 10 inode 및 directory 구조체 수정

2022. 12. 11 ~ 2022. 12. 13 process.c와 syscall.c 수정

2022. 12. 14 ~ 2022. 12. 16 buffer cache와 directory 시스템 구현

2022. 12. 17 ~ 2022. 12. 19 inode를 index structure로 구현

2022. 12. 20 ~ 2022. 12. 22 filesys와 inode 완성 및 디버깅

2022. 12. 23 ~ 2022. 12. 25 보고서 작성 및 오류 수정

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* Extensible file & file growth

inode.h와 inode.c에서 먼저 on file structure 중 하나인 inode와 함께 inode에 대응되는 index block인 inode\_disk 구조체를 선언하는데, block\_sector\_t type을 지니는 direct\_blocks, indirect\_block, doubly\_indirect\_block 멤버 변수를 추가한다. 또한 각 indirect block 마다 대응될 수 있는 block 구조체인 inode\_indirect\_block을 정의한다.

파일의 크기는 inode\_disk의 length 멤버 변수로 추가하여 관리한다. byte\_to\_sector 함수에서 direct block, single indirect block, indirect block에서 현재 접근하고자 하는 byte가 파일의 어떠한 block sector에 속하는지를 찾을 수 있도록 하는데, 이를 위해 수업 시간에 배운 index block의 파일 위치 구하는 식을 사용하여 해당 byte가 어떠한 block sector number에 대응되는지를 구한다. 만약 doubly indirect block이면 block을 두 번 따라갈 수 있도록 index1과 index2를 통해 두 번 구한다.

파일을 생성할 때는 처음 inode도 같이 생성하면서 inode\_create 함수가 실행되는데, 이때 inode\_disk 구조체의 내용도 같이 미리 할당되도록 sector\_allocate 함수를 추가로 구현하여 사용한다. 여기서도 현재 파일이 가리키는 포인터 pos가 어떠한 block sector를 가리키는지를 구하여 inode\_disk의 direct 또는 indirect block과 연결해준다. 또한 파일을 삭제할 때도 inode\_disk가 가리키는 block을 모두 삭제해야 하므로 sector\_deallocate 함수를 통해 for loop으로 탐색할 수 있도록 구현했다.

* Subdirectory

directory.c의 dir\_create 함수에서 inode\_create 함수를 호출할 때 현재 생성하고자 하는 inode가 디렉토리에 대응된다는 것을 업데이트 하기 위해 true 매개변수를 추가한다. dir\_remove 함수에서는 현재 디렉토리를 나타내는 ‘.’ 또는 부모 디렉토리를 의미하는 ‘..’가 file의 name으로 오면 해당 디렉토리를 삭제하지 못하도록 하는 코드를 조건문으로 추가한다. 또한 dir\_readdir로 name 이름을 지닌 디렉토리의 내용을 읽을 때 ‘.’ 또는 ‘..’가 나오면 탐색을 skip하도록 한다.

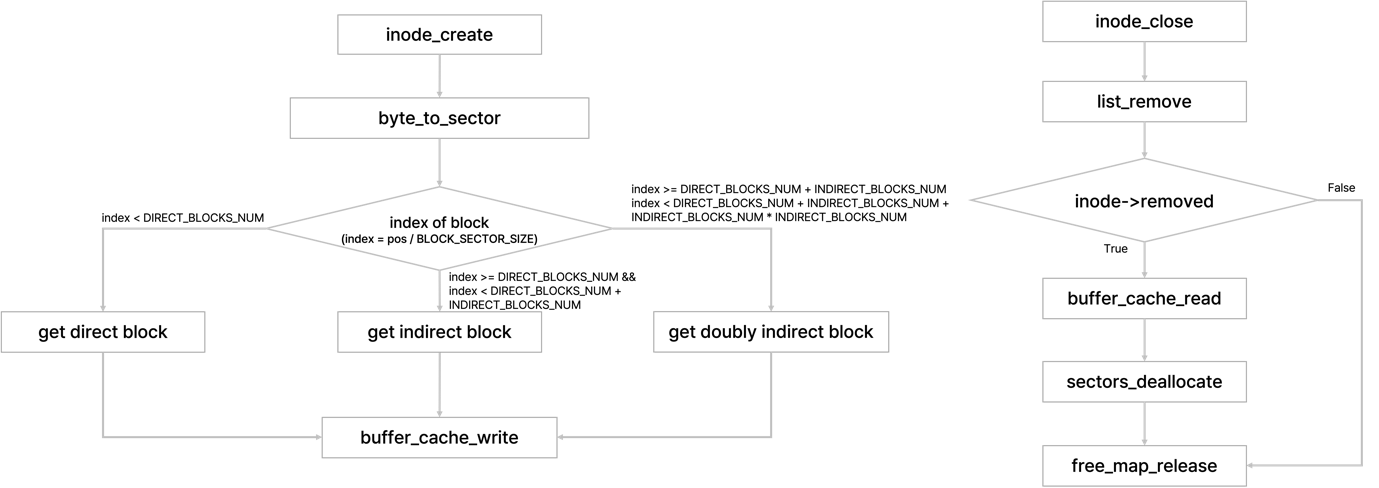
디렉토리 또는 파일 경로의 최대 길이 PATH\_MAX는 256으로 설정했고, 경로가 주어졌을 때 가장 하위에 있는 디렉토리 또는 파일을 parsing 할 수 있도록 split\_path 함수를 추가했다. filesys\_init 함수에서는 buffer\_cache\_init 함수를 호출하고 현재 thread의 cur\_dir이 dir\_open\_root 함수로 새롭게 연 디렉토리로 할당했다. filesys\_create 함수에서는 name 경로를 가진 파일을 새로 생성하는 것인데, 기존 PintOS 코드에서 success 변수에 결과를 할당하는 자리에 dir\_add의 파라미터에서 단순히 같은 파라미터 name을 넘기지 않고 split\_path 함수로 마지막 디렉토리 이름을 parsing한 결과를 넘겼다. 마찬가지로 filesys\_open 함수와 filesys\_remove 함수에서도 경로가 아닌 파일 이름으로 관리되는 파일을 삭제할 수 있도록 dir\_lookup 함수와 dir\_remove 함수의 파라미터에 parsing한 이름을 넘기도록 고쳤다. do\_format 함수에서는 기본적으로 비어있는 root directory에 ‘.’와 ‘..’는 기본적으로 내용으로 담기도록 dir\_add 함수를 호출한다. syscall에서 새롭게 만들어야 하는 시스템 콜인 chdir와 mkdir 명령어에 대응될 수 있는 filesys\_mkdir 함수와 filesys\_chdir 함수를 filesys\_create와 같은 방법으로 추가로 구현했다.

* Buffer cache

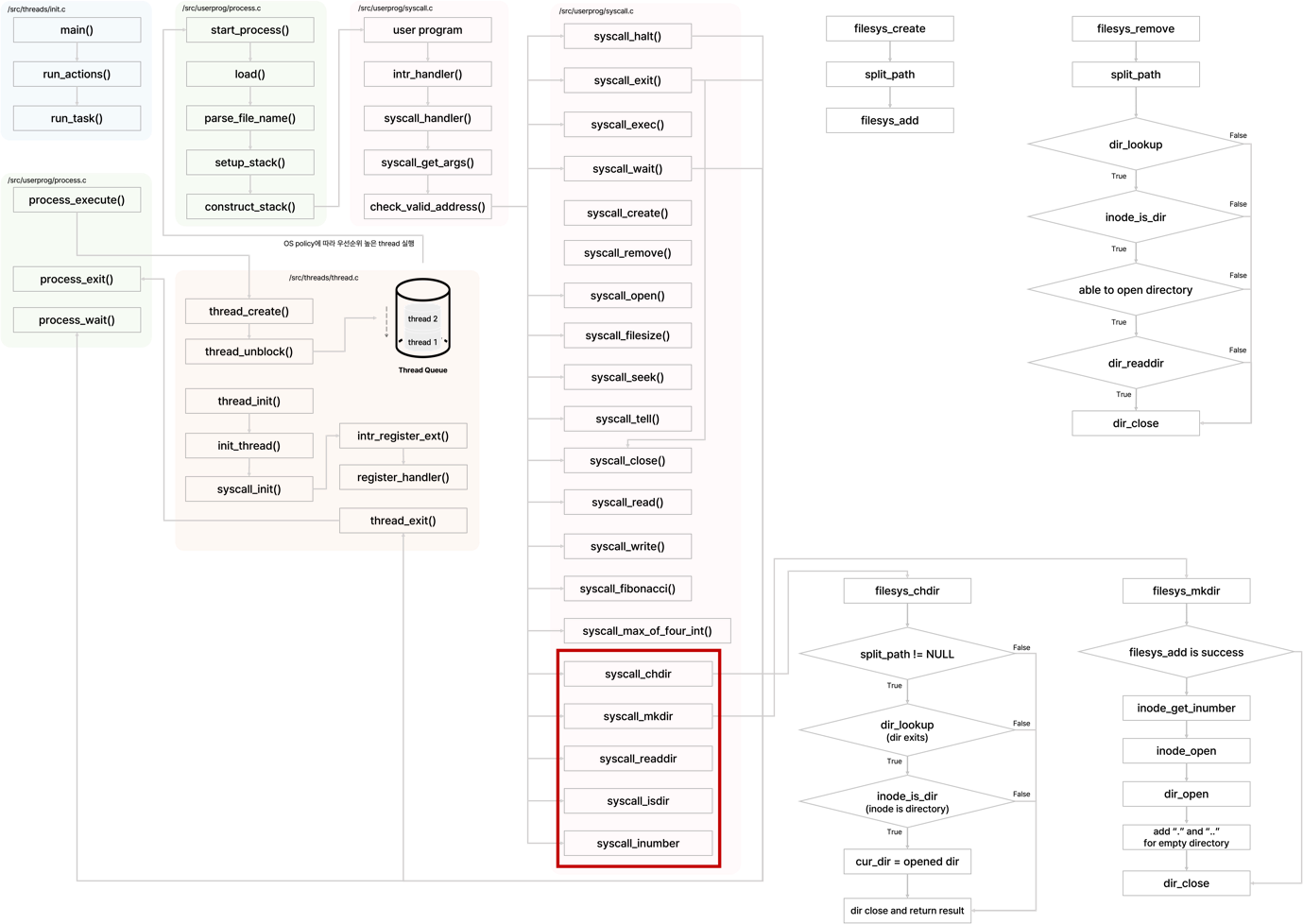
Buffer cache의 entry를 for loop으로 초기화하는 buffer\_cache\_init, 파일 시스템을 종료해야 하는 상황에서 buffer의 모든 내용을 파일로 flush 하기 위한 buffer\_cache\_terminate, 파일의 sector에 대응되는 buffer cache entry를 찾아서 데이터를 memcpy 함수로 읽거나 쓰는 buffer\_cache\_read와 buffer\_cache\_write, 모든 또는 어떤 특정 buffer cache entry를 메모리에서 데이터로 block\_write 함수를 통해 flush하는 buffer\_cache\_flush\_all과 buffer\_cache\_flush\_entry 함수를 추가했다. 또한 buffer\_cache\_write 함수를 구현할 때 현재 block sector에 대응되는 buffer cache entry가 없으면 victim entry를 찾아서 디스크에서 메모리로 가져온 후 써야 하는데, 이때 victim entry를 고르는 방법으로써 clock algorithm으로 구현한 buffer\_cache\_select\_victim\_by\_clock과 LRU algorithm으로 구현한 buffer\_cache\_select\_victim\_by\_lru 함수를 추가했다. 실제로 최종 제출 때는 LRU 알고리즘을 택했으며, 이를 위해 buffer\_cache\_entry 구조체에 access\_time이라는 멤버변수를 추가했다. 매번 buffer cache entry에 접근할 때 tick 함수로 현재 CPU tick을 가져와 저장할 수 있도록 했다. 또한 각 buffer cache entry를 접근하는 데 있어서 mutual exclusion이 보장되도록 buffer\_cache\_entry에 lock을 추가하여 entry 접근 전과 후에 lock을 얻거나 해제할 수 있도록 구현했다.

1. **연구 결과**
   1. **Flow Chart**

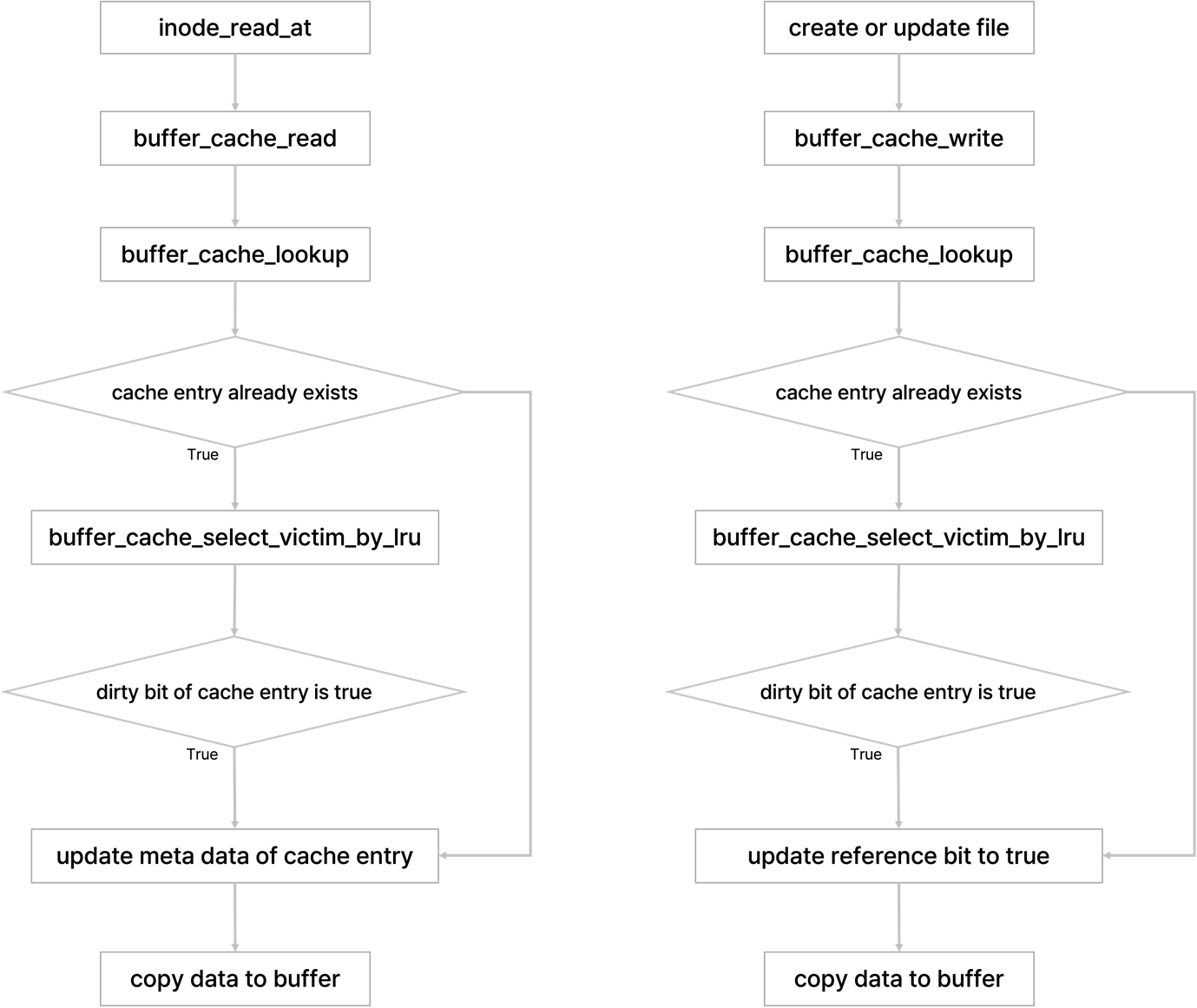
* **II. B. 개발 내용에 대한 Flow Chart를 작성**
* Extensible file & file growth



* Subdirectory



* Buffer cache



* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**
* Extensible file & file growth

텍스트이(가) 표시된 사진

자동 생성된 설명

In-memory 구조인 inode\_disk에 direct\_blocks 배열과 indirect\_block, 그리고 doubly indirect\_block 변수를 추가한다. 또한 inode에 있는 indirect\_block에 대응시킬 수 있는 blocks 배열을 지닌 inode\_indirect\_block 구조체를 추가한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

파일 시스템은 기본적으로 block(sector) 단위로 관리되므로 어떤 바이트 위치가 왔을 때 이 바이트가 위치한 block의 번호를 구할 수 있어야 한다. 이를 위해 byte\_to\_sector 함수를 구현했다. 파라미터의 pos가 현재 바이트의 위치를 가리키고, 해당 바이트 위치가 direct block에 있는지, singly indirect 또는 doubly indirect block에 있는지 확인해야 한다. 이는 pos를 block의 크기로 나눈 값인 index가 어떠한 범위에 해당되는지를 조건문으로 분기하여 처리하는데, pos가 direct block 개수보다 적으면 direct blocks 배열의 원소를 반환한다. direct block 개수보다 index가 크면 singly indirect block 개수보다 작은지를 확인한다. 작으면 현재 pos 바이트는 singly indirect block에 위치하는 것이므로 index에서 direct block의 개수를 뺀 결과를 index로 해서 indirect block 배열 원소를 반환한다. 만약 이 범위에도 들어오지 않으면 doubly indirect block이라는 뜻이므로 다음과 같은 식으로 두 번 block을 따라가서 대응되는 원소를 access한다.

첫 번째 access하는 indirect block의 index = (index – (direct block 수) – (indirect block 수)) / (indirect block 수)

두 번째 access하는 indirect block의 index = (index – (direct block 수) – (indirect block 수)) % (indirect block 수)

block 번호를 찾으면 메모리의 buffer cache에 block에 있는 내용을 buffer\_cache\_read를 통해 읽어오고, 대응되는 block이 없으면 -1을 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

inode\_create에서는 디스크의 파일에 관한 정보를 지니는 Inode를 생성하는 함수이다. 먼저 inode는 INDIRECT\_BLOCKS\_NUM 개수만큼 block을 지니고 있으므로 이 block들을 모두 -1로 초기화하고, 파일의 첫 위치부터 block 번호를 하나씩 늘려가면서 block\_sector를 free\_map의 비트를 1로 할당하고 sector 할당이 정상적으로 되면 buffer\_cache\_write를 호출하여 0로 cache entry를 초기화하고, block 할당이 모두 마무리 되면 buffer\_cache\_write로 호출하여 disk\_inode인 파일의 메타 정보를 메모리 cache entry에 쓴다.

텍스트이(가) 표시된 사진

자동 생성된 설명

생성한 inode를 close하면서 디스크로 해당 inode 내용을 써주는 역할을 담당하는 inode\_close 함수에서는 inode가 삭제된 경우, 다시 말해 inode의 removed 변수가 true이면 디스크에 buffer cache entry 내용을 써 주면서 sectors\_deallocate 함수를 호출하여 메모리를 해제한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

inode\_read\_at 함수에서는 buffer\_cache\_read 함수 호출을 통해 파일의 inode에 해당되는 buffer cache entry의 내용을 읽어서 파일의 메타 정보를 추출하고, 이를 바탕으로 파일의 block 번호 내용이 가리키는 부분의 메모리 데이터를 읽어서 buffer 파라미터에 가져온다.

텍스트이(가) 표시된 사진

자동 생성된 설명

inode\_write\_at 함수에서는 먼저 파일의 메타 정보가 있는 inode\_disk의 내용을 buffer cache entry에서 읽어오고, 현재 파일에서 쓰려는 위치인 offset부터 size만큼 크기를 쓸 수 있는 공간이 할당되어 있는지 확인해서 공간이 충분하지 않으면 block을 offset에서 size 크기만큼 block을 할당한다. 파일 크기를 늘렸으므로 그에 맞게 inode\_disk의 buffer cache entry 내용도 업데이트한다. 이후 파라미터 buffer에 있는 내용을 buffer\_cache\_write로 buffer cache entry에 써 준다.

* Subdirectory

텍스트이(가) 표시된 사진

자동 생성된 설명

split\_path는 파일 또는 디렉토리의 경로가 절대 경로인지 상대 경로인지 확인하고, ‘/’로 구분하여 파일 탐색을 진행한다. 디렉토리면 그 자체를 반환하고 파일이면 NULL을 반환한다. 그러면 parsing된 디렉토리 또는 파일 이름을 얻을 수 있다.이를 ‘/’를 구분자로 tokenize하여 차례대로 탐색하고, 다음 token이 NULL일 때 strlcpy를 통해 파일 이름을 가져온다. token과 next\_token을 같이 사용하는 이유는 next\_token은 현재 보고자 하는 파일 또는 디렉토리이고, token은 그 파일 또는 디렉토리가 위치한 디렉토리이다. 이 함수를 모두 실행하고 나서 token으로 찾은 현재 위치한 디렉토리를 반환해서 이 디렉토리를 dir\_lookup으로 접근하여 해당 파일을 읽을 수 있도록 하기 위함이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

filesys\_init에서는 buffer\_cache\_init을 호출하고 현재 thread가 참조하는 디렉토리인 cur\_dir가 root 디렉토리가 되도록 dir\_open\_root 함수를 호출한 반환값을 할당한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

생성한 파일을 free\_map에 1 비트로 업데이트하고, 업데이트가 가능하면 파일일 경우 inode 생성을 위해 inode\_create를, 디렉토리일 경우 dir\_create를 호출한다. 이후 현재 디렉토리에 파일 또는 디렉토리를 더하고 만약 이 추가 작업이 제대로 되지 않을 경우 free\_map에서 0 비트로 초기화한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

filesys\_create 함수로 파일 또는 디렉토리를 생성할 때 split\_path 함수와 filesys\_add 함수를 차례로 호출하여 파일 또는 디렉토리를 생성한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

filesys\_open에서는 파라미터 name을 경로로 지닌 파일 또는 디렉토리를 file\_open 함수를 사용하여 열기 위한 함수로 split\_path 함수를 사용하여 해당 파일 또는 디렉토리가 위치한 파일 위치를 연 다음, dir\_lookup 함수를 호출하여 해당 디렉토리에 접근해 파일 또는 디렉토리의 inode를 확인한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

filesys\_remove 함수로 파일 또는 디렉토리를 삭제할 때는 먼저 삭제하고자 하는 파일이 실제로 dir 디렉토리에 존재하는지 확인하고, 파일의 inode를 참조하여 해당 파일이 디렉토리이면 해당 파일을 읽어서 파일을 삭제할 수 있으면 dir\_remove 함수를 호출하여 삭제한다.



filesys\_chdir는 system call에서 디렉토리를 변경하는 syscall\_chdir를 호출할 때 실행되는 함수이며, parsing한 파일 이름이 실제로 디렉토리인지 확인하고 디렉토리이면 dir\_close 함수를 호출하여 현재 디렉토리를 닫고 parsing한 디렉토리를 새로 thread의 cur\_dir에 할당한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

filesys\_mkdir는 system call에서 디렉토리를 생성하는 syscall\_mkdir를 호출할 때 실행되는 함수이며, split\_path 함수를 통해 디렉토리 이름을 parsing하고 디렉토리를 파일 시스템에 filesys\_add 함수로 생성하여 이것이 성공하면 빈 디렉토리를 만든 것이다. 그런데 빈 디렉토리라고 해서 directory entry에 아무것도 없어야 하지는 않고 현재 디렉토리를 의미하는 ‘.’와 부모 디렉토리를 뜻하는 ‘..’를 entry로 할당해줘야 하므로 dir\_add 함수를 호출하여 entry를 추가한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

write\_bad\_ptr에서 계속 무한 로딩이 되는 failed 에러가 나서 이를 해결하기 위해 if 조건문에 pagedir\_get\_page 함수 호출을 추가해 현재 thread가 접근하려는 주소의 페이지가 존재하는지를 확인하도록 했다.

* Buffer cache  
  **구현한 경우만 작성**

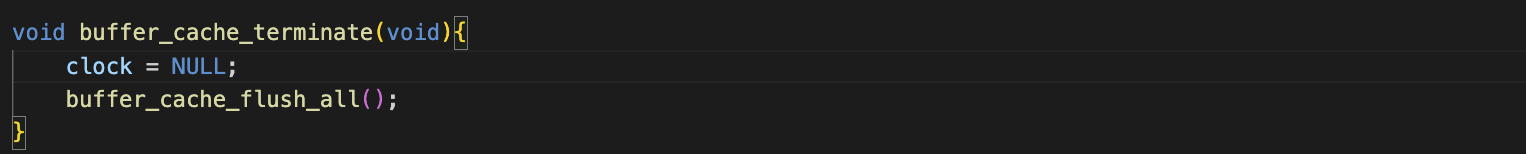


buffer\_cache\_entry 구조체에 buffer cache의 메타 데이터와 실제 cache에 저장될 수 있는 block sector 크기만큼의 buffer 배열을 구현했다. valid\_bit은 cache entry가 접근 또는 작성이 유효한지, dirty\_bit은 cache entry의 내용이 이전에 한 번 수정된 적이 있는지, reference\_bit은 clock algorithm에서 각 entry가 second chance를 지니고 있는지에 관한 비트이다. Synchronization을 보장하기 위해 mutex lock을 추가했고, 최종적으로 LRU 알고리즘을 eviction으로 사용했으므로 access\_time 변수를 추가하여 각 cache entry가 어떤 tick에 접근되었는지를 저장한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

buffer\_cache\_init 함수에서는 처음 buffer cache system을 시작할 때 각 cache entry의 메타 데이터와 lock을 초기화한다. Clock algorithm에서 처음 시작하는 clock pointer가 cache entry의 첫 번째 원소를 가리키도록 한다.



buffer\_cache\_terminate 함수에서는 clock pointer를 NULL로 초기화하고 buffer cache에 있는 모든 내용을 디스크에 쓰게 하는 buffer\_cache\_flush\_all 함수를 호출한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

buffer\_cache\_lookup 함수를 사용하여 block sector 번호에 대응되는 cache entry가 존재하는지를 탐색한다. for loop으로 entry의 disk\_sector 값이 찾고자 하는 disk\_sector 값과 같은지 확인하고, valid\_bit까지 1이면 해당 entry의 reference\_bit을 1로 설정한 후 entry 자체를 반환한다. cache entry가 존재하지 않으면 NULL을 반환한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

buffer\_cache\_select\_victim\_by\_lru 함수는 앞서 언급한 LRU 알고리즘으로 eviction할 victim entry를 선정하는 함수이다. for loop으로 cache 배열을 탐색하면서 이제까지 얻은 가장 작은 min\_access\_time보다 더 작은 access\_time이 나오면 현재 보고 있는 entry를 victim으로 선정하고 min\_accesss\_time도 해당 cache entry의 access\_time으로 업데이트한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

buffer\_cache\_flush\_entry 함수는 cache\_entry의 buffer 내용을 실제 디스크의 block에 작성하는 역할을 한다. 그래서 block\_write 함수를 사용한 후, 해당 cache\_entry의 내용을 모두 초기화해야 하므로 memset으로 buffer 내용을 0으로 초기화하고 메타 데이터도 모두 0으로 설정한다.

buffer\_cache\_flush\_all 함수는 cache 배열의 모든 entry 내용을 디스크에 작성하는 것으로, cache 배열을 for loop을 돌면서 buffer\_cache\_flush\_entry 함수를 호출하여 해당 배열의 원소인 cache\_entry의 buffer 내용을 디스크에 옮겨 작성한다.

이 과정에서 synchronization이 보장될 수 있도록 앞뒤에 각각 buffer\_cache\_lock과 각 cache entry의 lock을 사용했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

buffer\_cache\_read 함수에서는 block sector에 대응되는 cache entry에서 파라미터인 buffer로 chunk\_size만큼 내용을 읽어오는 일을 수행한다. 먼저 block sector 번호에 대응되는 cache entry가 존재하는지 buffer\_cache\_lookup 함수를 호출하여 확인하고, entry를 찾을 수 있으면 reference bit을 1로 설정하고 access\_time을 timer\_tick 함수의 반환값으로 업데이트한 후 memcpy 함수를 사용하여 buffer에 cache\_entry의 내용을 복사하여 읽어온다. 만약 찾고자 하는 block sector에 대응되는 cache entry가 존재하지 않을 경우 LRU 또는 clock algorithm으로 victim entry를 정하고 만약 dirty\_bit이 1이면 buffer\_cache\_flush\_entry 함수를 호출하여 해당 entry의 내용을 디스크에 업데이트한다. 이후 cache entry의 메타 데이터를 업데이트하는데, LRU 알고리즘을 위해 access\_time에 현재 CPU의 tick인 timer\_ticks 함수의 반환값을 할당하여 counter 기반의 LRU 알고리즘을 구현할 수 있도록 했다. 그후 마찬가지로 memcpy 함수를 통해 buffer에 cache entry의 내용을 복사하여 읽어온다.

텍스트이(가) 표시된 사진

자동 생성된 설명

buffer\_cache\_write 함수에서는 buffer\_cache\_read 함수와 크게 다르지 않지만, block sector에 대응되는 cache entry에 파라미터인 buffer로 chunk\_size만큼 내용을 써 주는 일을 수행한다는 것이 차이다. 먼저 block sector 번호에 대응되는 cache entry가 존재하는지 buffer\_cache\_lookup 함수를 호출하여 확인하고, entry를 찾을 수 있으면 reference bit을 1로 설정하고 access\_time을 timer\_tick 함수의 반환값으로 업데이트한 후 memcpy 함수를 사용하여 buffer의 내용을 cache\_entry에 복사하여 써 준다. 또한 cache\_entry의 dirty\_bit도 1로 업데이트 한다는 점이 buffer\_cache\_read 함수와의 차이다. 만약 찾고자 하는 block sector에 대응되는 cache entry가 존재하지 않을 경우 LRU 또는 clock algorithm으로 victim entry를 정하고 만약 dirty\_bit이 1이면 buffer\_cache\_flush\_entry 함수를 호출하여 해당 entry의 내용을 디스크에 업데이트한다. 이후 cache entry의 메타 데이터를 업데이트하는데, LRU 알고리즘을 위해 access\_time에 현재 CPU의 tick인 timer\_ticks 함수의 반환값을 할당하여 counter 기반의 LRU 알고리즘을 구현할 수 있도록 했다. 그리고 현재 buffer cache에는 block 내용이 없으므로 block\_read 함수를 호출하여 실제 디스크에서 block 내용을 읽어와 cache의 buffer에 저장한다. 그후 마찬가지로 memcpy 함수를 통해 buffer의 내용을 복사하여 cache entry의 내용에 써 주고 dirty\_bit을 1로 설정해 준다.

* 1. **시험 및 평가 내용**
* **Src/filesys make grade 수행결과를 캡처 하여 첨부.**

텍스트이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명