**Pintos Project 5: Filesystem**

담당 교수 : 김영재 교수

조 / 조원 : 20171682 임정호

개발 기간 : ’21.12.01 ~ ’21.12.22

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

1. buffer cache를 구현하여 file system에 적용한다.
2. external fragmentation을 해결하여 저장할 수 있는 파일의 크기와 크기를 늘릴 수 있게 한다.
3. Subdirectory를 지원할 수 있게 한다.

**2. 개발 범위 및 내용**

* 1. **개발 범위**
* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Extensible file & file growth

기존의 pintos에서 파일은 연속된 sector들로 이루어져 있었기 때문에, 파일의 크기를 늘리거나, 큰 파일을 저장하기 곤란하였다. Extensible file과 file growth를 구현하여, external fragmentation 문제를 해결하면 파일의 크기를 늘릴 수 있고, 최대 8MB 크기의 파일을 저장할 수 있다.

1. Subdirectory

기존의 pintos는 root directory만 지원하는 구조였지만, subdirectory를 지원할 수 있다. 절대 경로나 상대경로에 상관없이 기존의 파일 시스템과 잘 통합되어 작동될 수 있다.

1. Buffer cache  
   **구현한 경우만 작성**

파일 시스템을 수정하여 file block들의 cache를 구현한다. cache를 구현하여, 매번 데이터 접근을 위해 disk 작업을 수행하지 않고, caching된 데이터를 사용할 수 있다.

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

external fragmentaion과 파일의 크기를 늘릴 수 있도록, 기존의 pintos file system에서 사용하던 single extent 방식이 아닌, double-indexing 방식을 사용한다. pintos에서 사용하던 inode disk에는 데이터가 저장된 disk 블록의 번호와 데이터가 얼만큼 저장되어 있는지 알려주는 length변수가 있었는데, struct inode\_disk 구조체를 수정하여 block들을 direct하게 저장하는 block\_sector\_t block과 double indexing을 지원하기 위한, indirect 블록과, doubly\_indirect 블록을 선언한다.

single extent방식에서 double-indexing 방법으로 수정했기 때문에, inode를 생성하고 삭제하는 방법 또한 다르다. 생성할 때는 저장할 파일이 필요로 하는 디스트의 sector개수를 파악하여, direct 블록에 저장할 수 있으면 direct블록에 저장하고, 필요한 sector의 개수가 direct블록을 넘어선다면, single indirect 블록을 사용하여 저장한다. 역시 필요한 sector 개수가 single indirect 블록을 넘어선다면 doubley indirect 블록을 사용한다. direct 블록을 먼저 소비하면서 저장하는 것보다는 필요한 섹터의 크기를 파악하여 처음부터 direct 블록에 저장할지, single-indirect 블록, doubly indirect블록에 저장할 지 결정한 다음에 데이터를 저장한다면, 훨씬 효율적으로 inode 블록을 사용할 수 있을 거라 생각한다. inode를 삭제하는 방식또한 생성하는 방식과 동일하게 수행할 수 있다.

* Subdirectory
  + Directory entry 관리 방법

프로젝트 4까지는 thread에서 접근하는 file을 길이 128의 struct file의 배열로 관리했는데, file\_ds라는 구조체를 새로 선언하여, file과 file이 저장되어 있는 디렉토리를 저장할 수 있게 하였다. 추가로 struct thread 구조체의 struct dir\* cwd를 멤버 변수로 추가하여 현재 cwd를 저장할 수 있게 하였고, chdir에서 사용할 수 있다. 디렉토리의 첫 번재 directory entry는 “. .”에 해당하는 부모 디렉토리의 inode의 sector 번호를 저장하여, 입력된 경로가 상대 경로인지, 절대 경로인지 확인한 후, dir\_lookup()에서 현재 디렉토리인지, 부모 디렉토리인지, 그 외의 경우인지로 case를 나누어 처리하였다.

* Buffer cache
  + Buffer cache eviction 방식

buffer cache 구조체의 reference bit을 선언하여, 이를 활용하는 clock 알고리즘을 사용하였다. 반복문으로 buffer cache를 돌며 invalid하거나, reference bit이 false 인 경우, 해당 cache를 선택한다. 선택한 cache의 dirty bit이 set되어 있는 경우라면, buffer cache를 flush 한다.

* + Buffer cache flush 방식

flush되는 cache의 디스크 섹터 번호와 저장되어 있는 내용을 디스크에 저장한다. 저장한 후, 해당 cache의 dirty bit, reference bit, valid bit을 새로 초기화한다.

buffer cache는 여러 thread가 concurrent하게 접근할 수 있으므로, 모든 작업은lock()을 활용하여 concurrency한 접근을 방지한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**
* ‘21.12.01 ~ ’21.12.03 : 관련 이론 공부
* ’21.12.04 : buffer cache 구현
* ’21.12.05 ~ ’21.12.13 : extensible file & file growth 구현
* ’21.12.13 ~ ’21.12.21 : sub directory 구현 및, testcase 오류 잡기
  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

struct inode\_disk 구조체를 수정한다. 기존의 멤버변수로 있던 unused를 지우고, 구 공간을 direct[]와 indirect, doubly\_indirect로 대체한다. 기존의 inode를 allocate하고 deallocate하는 함수를 수정하는데, 저장할 파일이 필요한 sector 개수와 저장된 파일이 사용하는 sector 개수에 맞춰 수정한다. struct inode\_indirect 구조체를 선언하여 하나의 indirect 블록을 표현할 수 있게 한다. 또한, indirect 블록에 접근할 때, indirect 블록이 가리킬 수 있는 direct 블록, indirect 블록의 개수에 맞춰서 해당하는 block에 접근될 수 있게 인덱스 처리에 유의한다. double indexing 방법으로는 기존의 single extent 방식에서 처리하는 byte\_to\_sector()로는 올바른 sector를 리턴할 수 없으므로, 수정한다. inode를 생성하고, 삭제하는 기능과 관련된 함수는 대부분 수정하였다. indirect 블록의 allocate와 deallocate하는 기능의 함수를 새로 추가하였다.

* Subdirectory

struct dir이나, sturct dir\_entry는 수정한 것이 없다. thread의 구조체 멤버 변수에 directory를 위해 몇 가지 변수를 추가하고, 수정하였다. 우선 현재 process의 cwd를 저장하기 위한, struct dir\* cwd를 구조체 멤버 변수로 추가하였다. 그리고, 기존의 file을 관리하던 struct file fd[]을 수정하였는데, 새로운 구조체 struct file\_ds를 선언하여 파일과 디렉토를 모두 저장할 수 있게 하였고, thread 구조체멤버 변수를 struct file fd[]에서 struct file\_ds\* fd[]로 수정하였다. sub directory의 접근을 허용해야 하므로, 해당 경로가 상대 경로인지, 절대 경로인지 확인하고, ‘/’를 기준으로 parsing하여 파일과 디렉토리명을 구분하는 기능을 하는 함수가 필요하다. single root directory가 아닌 subdirectory이기 때문에, directory를 생성, 삭제 등의 처리를 할 수 있게, 기존의 dir\_XX()를 수정해야 한다. inode가 저장하는 데이터가 file인지 directory인지 구분하는 flag 변수도 inode 구조체에 추가하였다.

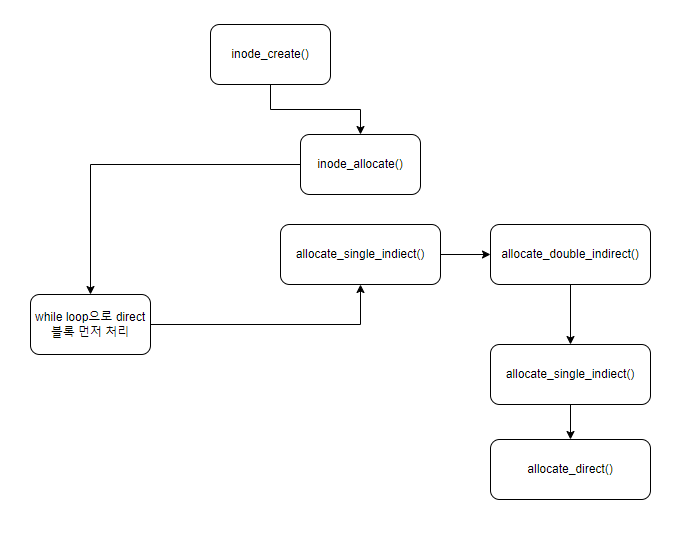
* Buffer cache

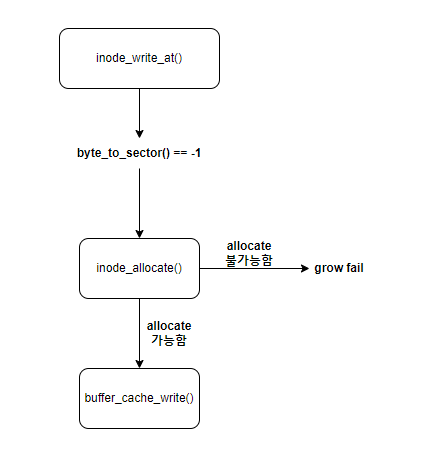
buffer cache를 나타내는 struct buffer\_cache\_entry구조체를 선언한다. buffer cache entry 배열의 buffer cache를 전역변수로 선언한 후, 이에 접근할 때 사용할 lock 변수도 선언한다. buffer cache와 lock 변수를 초기화하는 함수, buffer\_cache에서 읽는 함수와 쓰는 함수, buffer cache를 탐색하는 함수, evict 될 victim을 찾는 함수, victim을 disk로 flush하는 함수 등이 추가되어야 한다.

그리고 기존의 block에 read하고 write하는 inode.c의 block\_write(), block\_read()를 buffer\_cache\_write(), buffer\_cache\_read()로 대체해야 한다.

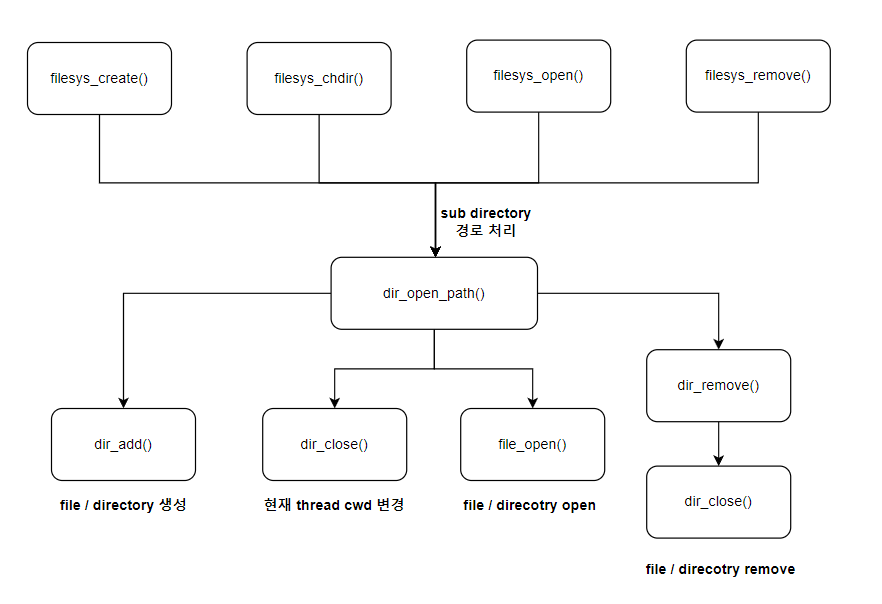
1. **연구 결과**
   1. **Flow Chart**

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

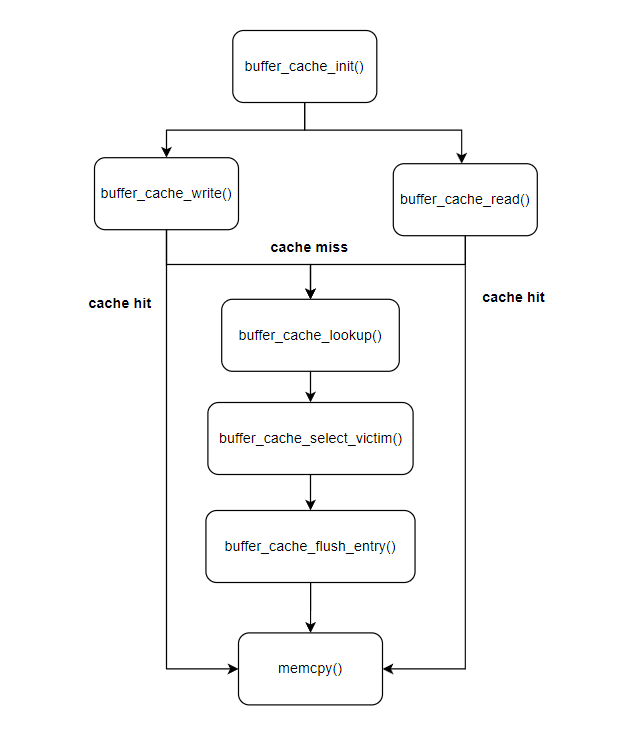




* Subdirectory

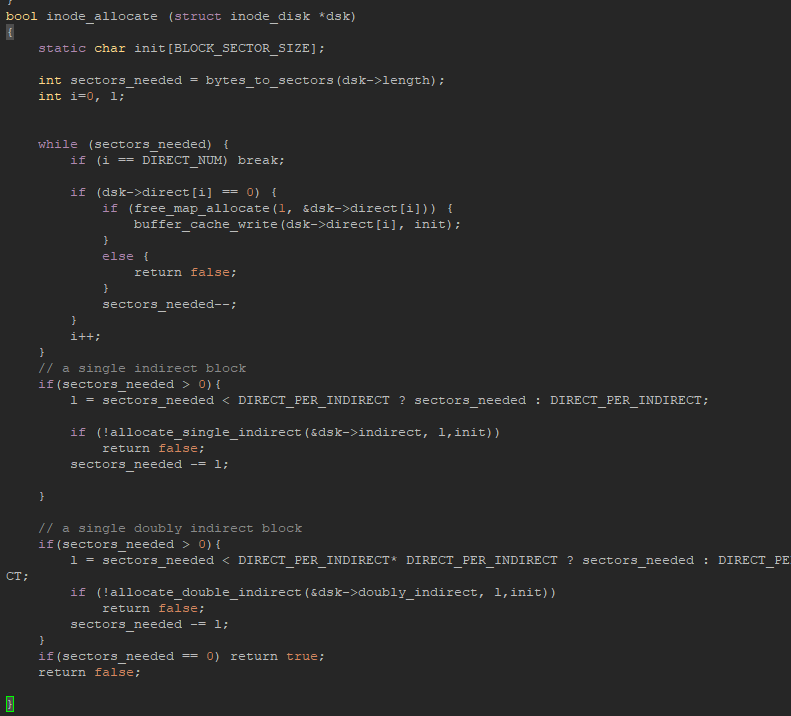


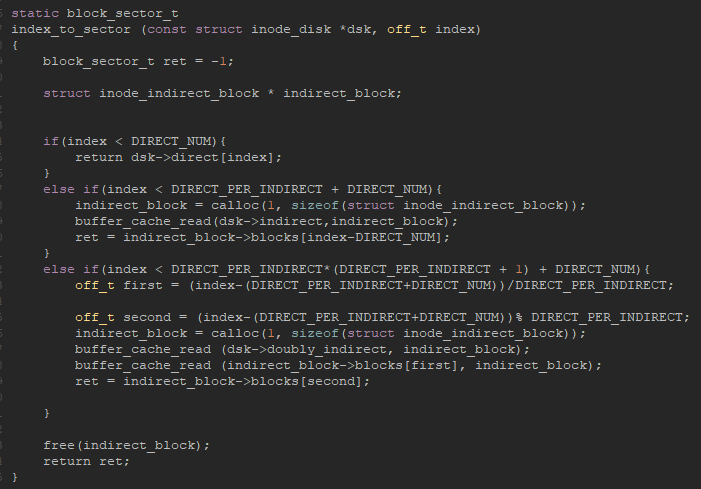
* Buffer cache  
  **구현한 경우만 작성**



* 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

기존의 inode\_disk 구조체에는 unused 배열이 존재했다. inode\_disk 구조체의 크기를 512로 맞춰주기 위해 존재했다고 판단하여 512바이트에 맞춰서 inode\_disk 구조체 를 변경했다. 길이 123짜리 block\_sector\_t형 배열 direct와 block\_sector\_t형 변수 indirect와 doubly\_indirect로 변경했다. indirect indexing을 지원하기 위한 자료구조 struct inode\_indirect 구조체를 선언하여 하나의 indirect 블록을 표현할 수 있게 한다. 하나의 indirect 블록이 가리킬 수 있는 block의 개수는 512/4인 128로 두었다. inode를 allocate하는 함수와 deallocate하는 함수를 수정했다. bytes\_to\_sectors()를 이용하여, 파일을 저장하기 위해 필요한 sector개수를 받아, direct 블록으로 해결할 수 있으면, direct 블록만 사용해서 파일을 저장하고, 파일의 크기가 direct 블록만으로 저장할 수 없으면 indirect, 같은 과정으로 doubly indirect 블록을 사용하여 allocate하게끔 함수를 구현하였다. allocate\_direct(), allocate\_single\_indirect(), allocate\_double\_indirect()의 함수를 구현하여, 기존의 file system에서 사용하던 inode\_allocate()에서 case를 나누어 inode를 allocate할 수 있게끔 하였다. deallocate()도 같은 과정으로 기존의 file\_system에서 사용하던 inode\_deallocate()에서 저장된 file이 사용하는 sector 개수만큼 case를 분류하여 deallocate\_direct(), deallocate\_indirect(), deallocate\_double\_indirect()를 구현하였다. double indexing을 이용하여 sector에 접근하기 때문에, doubly\_indirect에 indirect\_block을 읽고, indirect\_block의 block[i]에서 다시 indirect\_block을 읽고, indirect\_block의 blocks[]에 접근해야 한다.

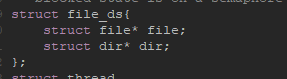






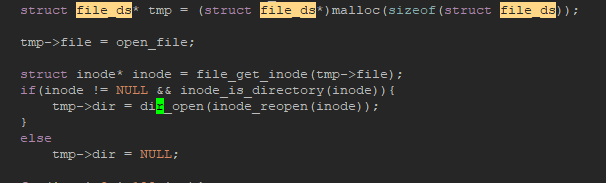
* Subdirectory

thread의 구조체 멤버 변수에 directory를 위해 몇 가지 변수를 추가하고, 수정하였다. 우선 현재 process의 cwd를 저장하기 위한, struct dir\* cwd를 구조체 멤버 변수로 추가하였다. 그리고, 기존의 file을 관리하던 struct file fd[]을 수정하였는데, 새로운 구조체 struct file\_ds를 선언하여 파일과 디렉토를 모두 저장할 수 있게 하였고, thread 구조체멤버 변수를 struct file fd[]에서 struct file\_ds\* fd[]로 수정하였다.

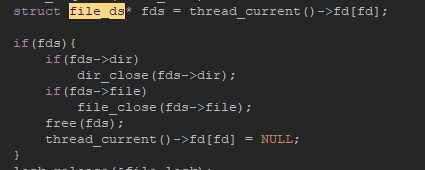
syscall.c에 있는 file system과 관련있는 대부분의 함수를 수정했다. open(), write(), read(), close(), tell(), seek()이다. file을 open할 때, 현재 open한는 file이 디렉토리이 경우라면 dir에 현재 디렉토리를 저장하고, 아닌 경우 NULL로 초기화한다.

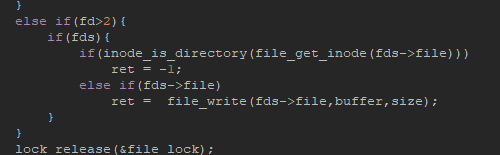
open()의 일부



file을 close할 때또한, file만 close하는 게 아니라, 해당 파일 디스크립터가 저장하던 디렉토리또한 close하고, thread의 fds를 메모리 해제 후, NULL로 초기화한다.

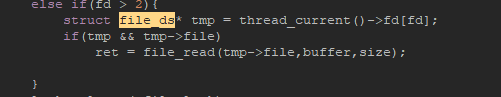
close()의 일부



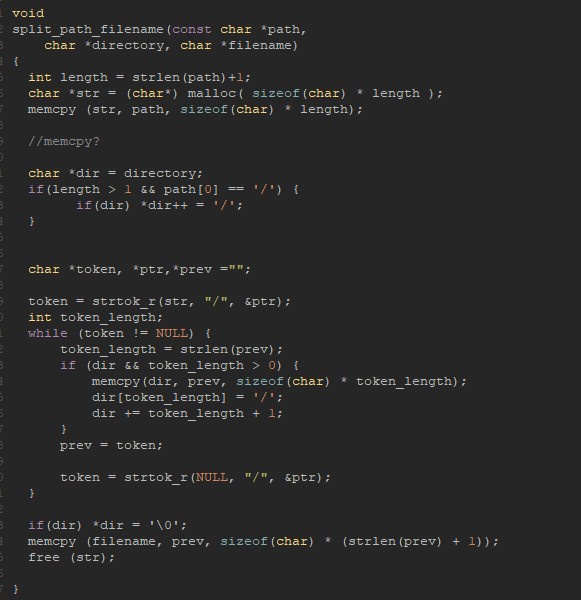
file을 read, write할 때 또한 마찬가지로, fds의 file로 접근하게끔 수정하였는데, write의 경우 write하려는 file이 디렉토리인 경우 -1을 리턴하게끔 하였더니, dir-open()문제가 pass되었다.

(왼) write()의 일부

(아래) read()의 일부



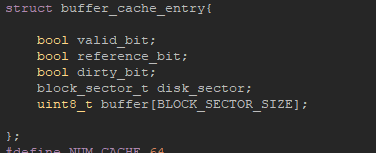
이번 프로젝트는 sub directory의 접근을 허용해야 하므로, 해당 경로가 상대 경로인지, 절대 경로인지 확인하고, ‘/’를 기준으로 parsing하여 파일과 디렉토리명을 구분하는 기능을 하는 함수가 필요하다. 어떠한 경로가 주어졌을 때, 디렉토리와 파일로 구분하는 함수와 주어진 경로의 디렉토리를 리턴하는 함수로 구분하여 구현하였고 다음과 같다.



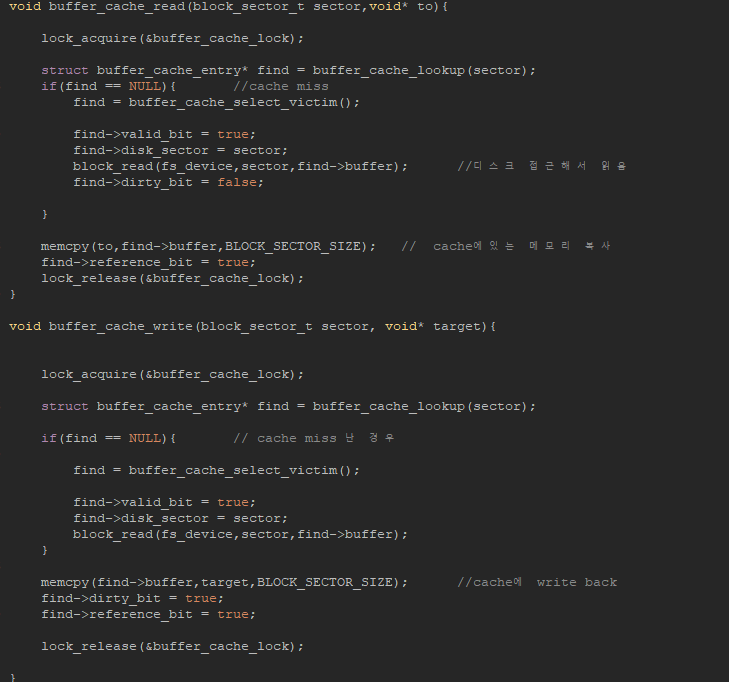


* Buffer cache  
  **구현한 경우만 작성**

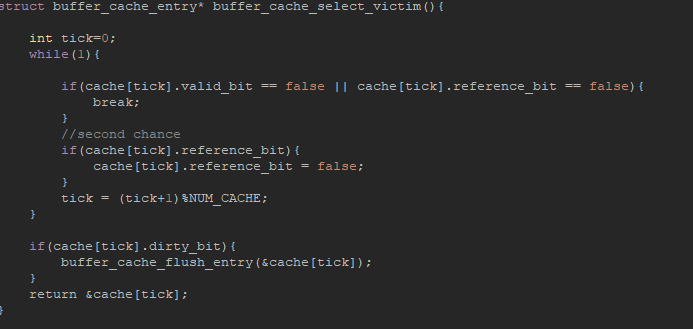
buffer\_cache\_entry 구조체를 선언하여, 그 멤버 변수로 해당 cache가 valid한지 나타내는 valid\_bit, evict의 기준의 될 reference bit, flush될 때, disk에 write을 해야 할 지 말지를 정하는 dirty\_bit등의 변수를 가진다.

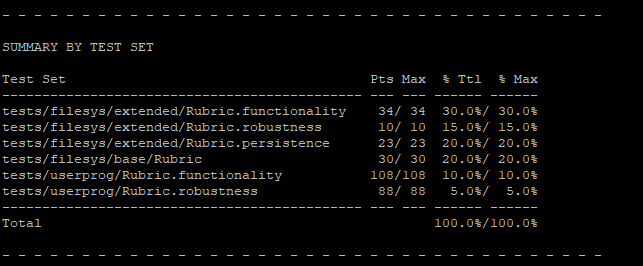


ppt 자료에서 제공한 구조체와, 함수 선언부를 바탕으로 구현하였는데, 모든 buffer cache의 초기 상태는 valid\_bit, reference\_bit, dirty\_bit 모두 false인 상태이다. 이 때, buffer\_cache를 read하거나 write하는 경우 reference\_bit가 true 초기화되고, 특히 write하는 경우에는 해당 buffer cache의 dirty\_bit가 true로 되어, buffer\_cache\_read()나 buffer\_cache\_write()가 호출되었을 때, cache miss가 발생한 경우 flush될 때, disk에 write될 수 있게 한다.



buffer\_cache\_read()나 buffer\_cache\_write()가 호출되어, cache를 찾고, cache hit인 경우라면 해당 cache에 저장되어 있는 데이터를 리턴하면 되지만, miss가 발생한 경우, cache에 새롭게 데이터를 수정해야한다. 만약 cache가 모두 꽉 차있는 경우라면, 어떠한 기준으로 기존에 있던 cache를 evict해야 한다. evict 알고리즘에는 여러 알고리즘이 있겠지만, ppt링크에서 설명했던 clock 알고리즘을 사용하였다. reference bit을 활용하는데, reference\_bit이 true로 되어 있으면, false로 변경하는 second chance를 제공한다. buffer cache를 돌면서, 사용되지 않는 cache이거나, reference\_bit이 false인 cache를 evict하고, 해당 cache에 disk에서 읽어온 데이터를 저장한다.



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