**Pintos Project 5: Filesystem**

담당 교수 : 박성용 교수

학번 / 이름 : 20181589 / 강민석

개발 기간 : 12/09 ~ 12/25

1. **개발 목표**

기존 파일 시스템은 파일을 지속적으로만 관리할 수 있기 때문에 섹터별로 구분하여 외부 단편화를 해결할 필요가 있다. 또한, 기존에는 고정된 파일 크기로는 file growth을 처리할 수 없었지만, 이를 해결하기 위해 file growth를 해결한다. 또한 기존에는 root directory만 존재했지만, 이를 확장하여 하위 디렉토리를 구현한다. 이를 위해 새로운 system call인 chdir, mkdir, readdir, isdir, inumber를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Extensible file & file growth

현재 pintos에서는 파일을 생성할 때에 파일의 크기가 결정되고 나서 추후에 변경이 불가능하다. 즉, file growth가 안된다. 그래서 이 항목에서는 파일에 write를 할 때 disk block을 할당 받아서 사용하도록 구현한다. 이를 통해서 파일 크기가 생성될 때 고정되지 않고 growth가 가능하도록 한다.

1. Subdirectory

현재 pintos에서는 root directory만 존재한다. 이런 단일계층 문제 때문에 root directory에만 파일을 생성할 수 있는데, 계층구조를 구현해서 하위 directory를 생성하고 거기에도 파일들이 생성될 수 있도록 이 문제를 해결한다.

1. Buffer cache

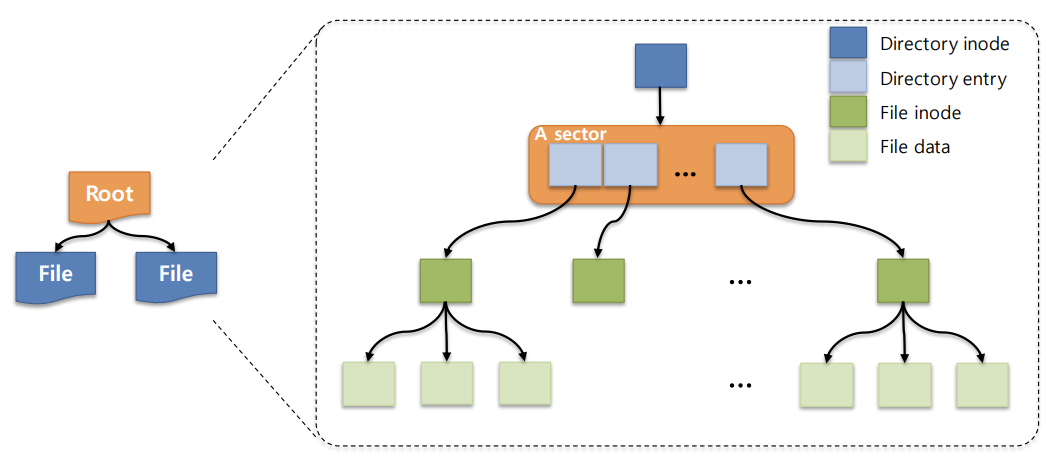
Buffer cache는 disk block을 캐싱하기 위한 메모리 영역이다. 디스크에서 입출력을 진행하려면 시간이 오래 걸리기 때문에, disk block을 메모리 영역에 둠으로써 파일의 입출력 응답시간을 줄인다. 현재 pintos에는 buffer cache가 존재하지 않기 때문에, 파일 입출력 할 때, 바로 디스크 입출력을 수행한다.

* 1. **개발 내용**
* Extensible file & file growth
  + Index structure와 management에 대해서 기술

Index Structure는 inode의 구조를 의미한다. Inode에서의 구조는 direct block, indirect block, 그리고 double indirect block으로 나뉘는데, direct block은 inode 자체가 block을 가리키는 것이고, indirect block은 inode가 또 다른 table을 가리키고 이 table의 entry가 block을 가리키는 것이다. Double indirect block은 두 단계로 간접적으로 접근을 하는 것이다. 결국 128개의 direct block index로 16384개의 sector를 가리킬 수 있어야 정상적으로 작동한다. 그래서 inode\_disk 구조체에 direct block index 개수만큼의 array를 구현한다.

* Subdirectory
  + Directory entry 관리 방법

기존의 pintos는 root directory 단 하나에 모든 파일이 존재하도록 구현되어 있다. 이것을 아래 그림과 같이, 한 디렉토리에 또 다른 디렉토리와 파일들이 존재할 수 있도록 subdirectory를 구현해야 한다.



각 thread마다 가지는 thread structure에 현재 디렉토리를 표현할 수 있는 directory pointer cur\_dir를 추가했고, 이를 통해서 디렉토리를 열고 닫는 등을 수행할 수 있도록 system call을 수정하였다. 또한, 디렉토리와 관련된 system call 다섯 개를 추가하였다. 디렉토리를 생성하는 mkdir, 현재 디렉토리에 해당하는 inode를 찾아내는 readdir, 입력된 경로의 디렉토리를 열고 이를 현재 스레드의 cwd로 바꿔주는 chdir, 디렉토리 여부를 반환하는 isdir, 그리고 디렉토리 식별 번호를 반환하는 inumber system call을 구현하였다.

* Buffer cache
  + Buffer cache eviction 방식 & Buffer cache flush 방식

Clock 알고리즘을 통해서 buffer cache에서 evict할 victim을 선택한다. 그리고 victim entry가 dirty일 경우 데이터를 디스크로 flush한다. 이를 구현하기 위해서 buffer\_cache\_entry 구조체에 dirty bit와 clock bit를 추가해 주었다. cache 전역변수의 entry를 순회하면서 clock bit 변수를 검사한다. 1일 경우에는 0으로 바꾸고, 0일 경우 victim으로 선택한다. 선택된 이후엔 dirty bit를 확인하여 dirty일 경우에는 디스크로 flush를 한다. 마지막으로 victim entry에 해당하는 cache 값을 업데이트하고 victim entry를 반환한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**
* 12/09 ~ 12/18: Buffer cache 및 Extensible file & file growth 구현
* 12/19 ~ 12/23: Subdirectory 구현
* 12/24 ~ 12/25: 마무리 및 보고서 작성
  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

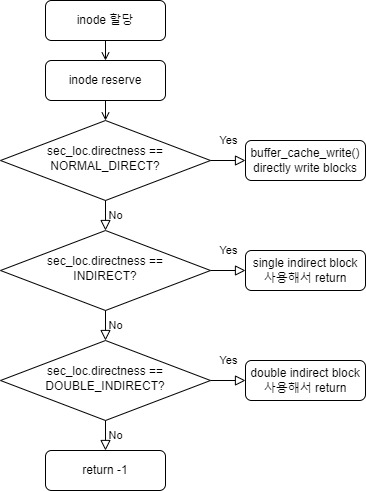
filesys/inode.c와 filesys/inode.h의 함수들을 조금씩 수정해서 구현한다.

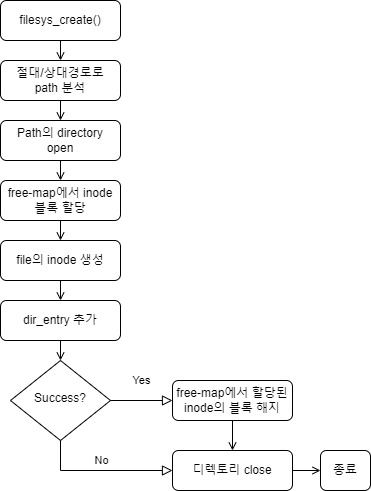
* + On-memory structure인 inode 구조체와 on-disk structure인 inode\_disk 구조체를 수정한다. 그리고 기존에 있던 inode.c의 함수들을 수정된 자료구조에 맞게 수정한다. 수정하는 함수는 byte\_to\_sector, inode\_create, inode\_open, inode\_read\_at, inode\_write\_at, 그리고 inode\_close이다.
  + static void locate\_byte(off\_t, struct sector\_location \*)
    - file offset을 통해 인덱스 블록의 offset 값을 계산한다.
  + static bool register\_sector(struct inode\_disk \*, block\_sector\_t, struct sector\_location)
    - 새로 할당 받은 디스크 블록의 physical address를 inode 자료구조에 갱신한다.
  + static bool inode\_update\_file\_length(struct inode\_disk \*, off\_t, off\_t)
    - file growth 시에 새로운 디스크 블록을 할당하고 inode를 업데이트한다.
  + static void free\_inode\_sectors(struct inode\_disk \*)
    - 파일에 할당된 모든 디스크 블록의 할당을 해지한다.
* Subdirectory
  + struct dir \*path\_parse(char \*name, char \*file\_name)
    - path name의 시작이 ‘/’ 여부에 따라 절대/상대 경로로 구분하여 디렉토리 정보를 dir에 저장한다. 그리고 strtok\_r() 함수를 이용해 path\_name의 디렉토리 정보와 파일 이름을 저장한다.
  + bool inode\_is\_dir(struct inode \*inode)
    - inde\_disk 자료구조를 메모리에 할당하여 in-memory inode의 on-disk inode를 읽어 inode\_disk에 저장한다. 그리고 on-disk inode의 is\_dir를 result에 저장하여 반환한다.
  + bool create\_dir(char \*name) (mkdir syscall에서 사용)
    - bitmap에서 inode sector 번호를 할당 받고, 해당 sector에 file\_name의 디렉토리를 생성한다. 그리고 디렉토리 엔트리에 file\_name entry를 추가하고 ‘.’ 파일의 엔트리를 추가한다.
  + bool change\_dir(char \*path) (chdir syscall에서 사용)
    - dir 경로를 분석하여 디렉토리를 반환한다. 그리고 스레드의 현재 작업 디렉토리를 변경한다.
  + int readdir(int fd, char \*name)
    - fd 리스트에서 fd에 대한 file 정보를 얻어오고, fd의 file->inode가 디렉토리인지 검사한다. 그리고 file pointer를 dir 자료구조로 가리키도록 하고 디텍토리 엔트리에서 “.” 이름을 제외한 파일 이름을 name에 저장한다.
  + filesys\_remove (const char \*name)
    - root 디렉터리에 파일 생성하는 대신 name 경로에 파일을 생성하도록 변경한다. 파일 생성 시, 절대/상대 경로를 분석해서 디렉토리 파일을 생성하도록 수정한다. 그리고 디렉토리 엔트리에서 file\_name의 in-memory inode가 파일인지 디렉토리인지 판단한다.
* Buffer cache

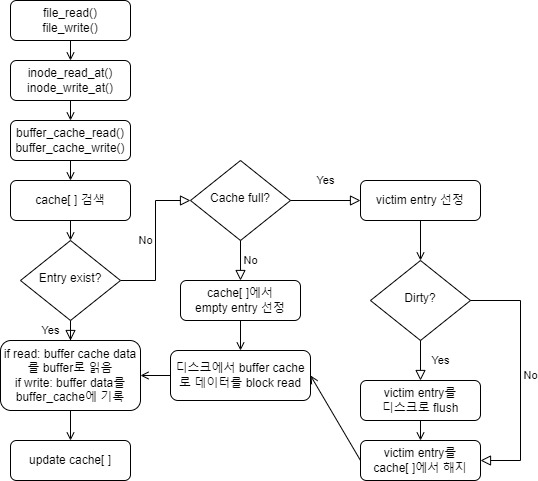
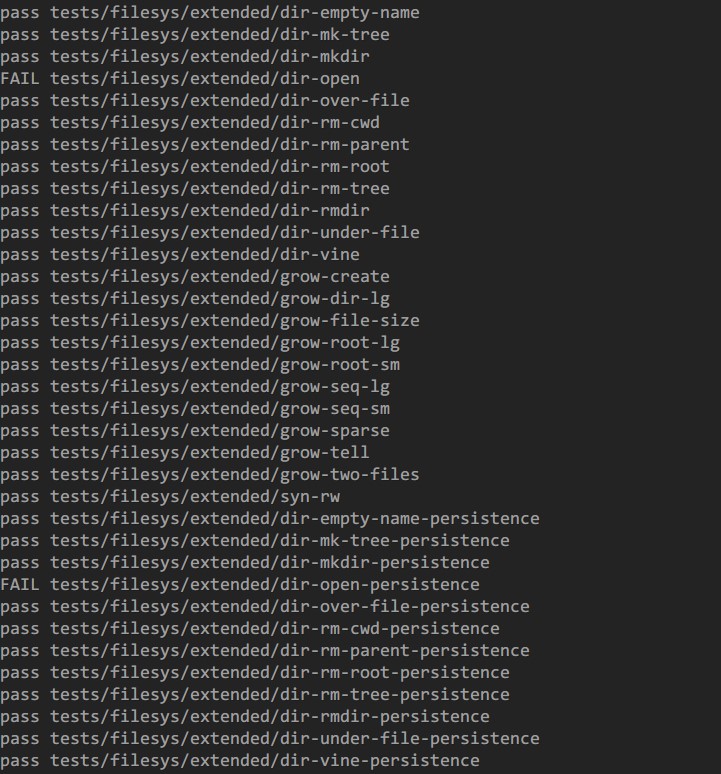
먼저, buffer cache를 가능하게 하기 위한 cache.c와 cache.h 파일을 filesys 폴더 내에 새로 만든다.

* + buffer\_cache\_entry 구조체 및 buffer\_cache\_entry cache[NUM\_CACHE] 배열
    - buffer cache 구현을 위해 꼭 필요한 구조체를 구현하고, entry 개수 만큼의 배열을 생성한다.
  + void buffer\_cache\_init(void)
    - buffer cache를 초기화하는 함수이다.
  + struct buffer\_cache\_entry \*buffer\_cache\_lookup(block\_sector\_t sector)
    - buffer cache를 순회하며 target sector가 존재하는지 검색한다.
  + bool buffer\_cache\_read(block\_sector\_t sector\_idx, void \*buffer, off\_t bytes\_read, int chunk\_size, int sector\_ofs)
    - buffer cache에서 요청 받은 buffer frame을 읽어온다.
  + struct buffer\_cache\_entry \*select\_victim(void)
    - Clock 알고리즘을 이용해 buffer cache에 자리가 없을 경우 buffer cache에서 victim을 선정하여 entry head 포인터를 반환한다.
  + void buffer\_cache\_flush\_entry(struct buffer\_cache\_entry \*b\_flush\_entry)
    - 인자로 주어진 entry의 dirty bit를 false로 설정하고, 해당 내역을 disk로 flush한다.
  + bool buffer\_cache\_write(block\_sector\_t sector\_idx, void \*buffer, off\_t bytes\_written, int chunk\_size, int sector\_ofs)
    - buffer cache의 buffer frame에 요청 받은 데이터를 기록한다.
  + void buffer\_cache\_terminate(void)
    - 모든 dirty entry를 flush하고 buffer cache를 해지한다.
  + void buffer\_cache\_flush\_all(void)
    - Buffer cache를 순회하며 dirty bit가 true인 entry를 모두 디스크로 flush한다.

1. **연구 결과**
   1. **Flow Chart**

* Extensible file & file growth
* Subdirectory



* Buffer cache
  1. **제작 내용**
* Extensible file & file growth
  + inode\_disk, inode structure 재구성
* struct inode\_disk
* {
* off\_t length; */\* File size in bytes. \*/*
* unsigned magic; */\* Magic number. \*/*
* block\_sector\_t direct\_map\_table[DIRECT\_BLOCK\_ENTRIES];
* block\_sector\_t indirect\_block\_sec;
* block\_sector\_t double\_indirect\_block\_sec;
* bool is\_dir;
* };
* struct inode
* {
* 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. \*/*
* struct lock extend\_lock;
* };
  + static void locate\_byte(off\_t, struct sector\_location \*): file offset을 통해 인덱스 블록의 offset 값을 계산한다.
* static void locate\_byte(off\_t pos, struct sector\_location \*sec\_loc)
* {
* off\_t changed\_pos = pos / BLOCK\_SECTOR\_SIZE;
* *if* (changed\_pos < DIRECT\_BLOCK\_ENTRIES)
* {
* sec\_loc->directness = NORMAL\_DIRECT;
* sec\_loc->index1 = changed\_pos;
* }
* *else* *if* (changed\_pos < INDIRECT\_BLOCK\_ENTRIES + DIRECT\_BLOCK\_ENTRIES)
* {
* changed\_pos -= DIRECT\_BLOCK\_ENTRIES;
* sec\_loc->directness = INDIRECT;
* sec\_loc->index1 = changed\_pos;
* }
* *else* *if* (changed\_pos < INDIRECT\_BLOCK\_ENTRIES \* (INDIRECT\_BLOCK\_ENTRIES + 1) + DIRECT\_BLOCK\_ENTRIES)
* {
* changed\_pos -= INDIRECT\_BLOCK\_ENTRIES + DIRECT\_BLOCK\_ENTRIES;
* sec\_loc->directness = DOUBLE\_INDIRECT;
* sec\_loc->index1 = changed\_pos / INDIRECT\_BLOCK\_ENTRIES;
* sec\_loc->index2 = changed\_pos % INDIRECT\_BLOCK\_ENTRIES;
* }
* *else*
* {
* sec\_loc->directness = OUT\_LIMIT;
* }
* }
  + static bool register\_sector(struct inode\_disk \*, block\_sector\_t, struct sector\_location): 새로 할당 받은 디스크 블록의 physical address를 inode 자료구조에 갱신한다.
* static bool register\_sector(struct inode\_disk \*inode\_disk, block\_sector\_t new\_sector, struct sector\_location sec\_loc)
* {
* int register\_flag = 0;
* struct inode\_indirect\_block block1, block2;
* block\_sector\_t \*sector;
* *if* (sec\_loc.directness == NORMAL\_DIRECT)
* {
* *if*(register\_flag == 0)
* {
* inode\_disk->direct\_map\_table[sec\_loc.index1] = new\_sector;
* *return* true;
* }
* }
* *else* *if* (sec\_loc.directness == INDIRECT)
* {
* register\_flag = 1;
* sector = &inode\_disk->indirect\_block\_sec;
* *for*(int i=0; i<10; i++)
* register\_flag++;
* *if* (\*sector == SECTOR\_MAGIC)
* {
* register\_flag = 0;
* *if* (free\_map\_allocate(1, sector))
* {
* register\_flag = 1;
* init\_indirect\_block(&block2);
* }
* *else*
* {
* register\_flag = 1;
* *return* false;
* }
* }
* *else*
* {
* register\_flag = 1;
* buffer\_cache\_read(\*sector, &block2, 0, sizeof(struct inode\_indirect\_block), 0);
* }
* *if* (block2.map\_table[sec\_loc.index1] == SECTOR\_MAGIC)
* {
* register\_flag++;
* block2.map\_table[sec\_loc.index1] = new\_sector;
* }
* *for*(int i=0; i<10; i++)
* register\_flag++;
* buffer\_cache\_write(\*sector, &block2, 0, sizeof(struct inode\_indirect\_block), 0);
* register\_flag = 0;
* *return* true;
* }
* *else* *if* (sec\_loc.directness == DOUBLE\_INDIRECT)
* {
* sector = &inode\_disk->double\_indirect\_block\_sec;
* register\_flag = 0;
* *if* (\*sector == SECTOR\_MAGIC)
* {
* *if* (free\_map\_allocate(1, sector))
* {
* init\_indirect\_block(&block1);
* }
* *else*
* {
* *return* false;
* }
* }
* *else*
* {
* buffer\_cache\_read(\*sector, &block1, 0, sizeof(struct inode\_indirect\_block), 0);
* }
* sector = &block1.map\_table[sec\_loc.index1];
* *for*(int i=0; i<10; i++)
* register\_flag++;
* *if* (\*sector == SECTOR\_MAGIC)
* {
* *if* (free\_map\_allocate(1, sector))
* {
* init\_indirect\_block(&block2);
* *if* (block2.map\_table[sec\_loc.index2] == SECTOR\_MAGIC)
* {
* block2.map\_table[sec\_loc.index2] = new\_sector;
* }
* register\_flag = 1;
* buffer\_cache\_write(inode\_disk->double\_indirect\_block\_sec, &block1, 0, sizeof(struct inode\_indirect\_block), 0);
* buffer\_cache\_write(\*sector, &block2, 0, sizeof(struct inode\_indirect\_block), 0);
* }
* *else*
* {
* register\_flag = 0;
* *return* false;
* }
* }
* *else*
* {
* buffer\_cache\_read(\*sector, &block2, 0, sizeof(struct inode\_indirect\_block), 0);
* *if* (block2.map\_table[sec\_loc.index2] == SECTOR\_MAGIC)
* {
* block2.map\_table[sec\_loc.index2] = new\_sector;
* }
* buffer\_cache\_write(\*sector, &block2, 0, sizeof(struct inode\_indirect\_block), 0);
* }
* *return* true;
* }
* *return* false;
* }
  + static bool inode\_update\_file\_length(struct inode\_disk \*, off\_t, off\_t): file growth 시에 새로운 디스크 블록을 할당하고 inode를 업데이트한다.
* static bool inode\_update\_file\_length(struct inode\_disk \*inode\_disk, off\_t start\_pos, off\_t end\_pos)
* {
* static char blankarray[BLOCK\_SECTOR\_SIZE];
* inode\_disk->length = end\_pos;
* int file\_length = 0;
* start\_pos = start\_pos / BLOCK\_SECTOR\_SIZE \* BLOCK\_SECTOR\_SIZE;
* end\_pos = (end\_pos - 1) / BLOCK\_SECTOR\_SIZE \* BLOCK\_SECTOR\_SIZE;
* *if*(file\_length != 0)
* {
* file\_length = 0;
* }
* *for* (;start\_pos <= end\_pos;)
* {
* file\_length++;
* block\_sector\_t find\_sector = byte\_to\_sector(inode\_disk, start\_pos);
* *if* (find\_sector != SECTOR\_MAGIC)
* {
* start\_pos += BLOCK\_SECTOR\_SIZE;
* *continue*;
* }
* *else*
* {
* *if* (free\_map\_allocate(1, &find\_sector))
* {
* file\_length++;
* struct sector\_location sec\_loc;
* locate\_byte(start\_pos, &sec\_loc);
* *if* (!register\_sector(inode\_disk, find\_sector, sec\_loc))
* {
* file\_length = 0;
* *return* false;
* }
* file\_length--;
* buffer\_cache\_write(find\_sector, blankarray, 0, BLOCK\_SECTOR\_SIZE, 0);
* start\_pos += BLOCK\_SECTOR\_SIZE;
* }
* *else*
* {
* file\_length = 0;
* *return* false;
* }
* }
* }
* *return* true;
* }
  + static void free\_inode\_sectors(struct inode\_disk \*): 파일에 할당된 모든 디스크 블록의 할당을 해지한다.
* static void free\_inode\_sectors(struct inode\_disk \*inode\_disk)
* {
* int i, j;
* int free\_done = 0;
* *if* (inode\_disk->double\_indirect\_block\_sec != SECTOR\_MAGIC)
* {
* struct inode\_indirect\_block \*first\_block\_ind = (struct inode\_indirect\_block \*)malloc(BLOCK\_SECTOR\_SIZE);
* buffer\_cache\_read(inode\_disk->double\_indirect\_block\_sec, first\_block\_ind, 0, sizeof(struct inode\_indirect\_block), 0);
* i = 0;
* free\_done ++;
* *while* (first\_block\_ind->map\_table[i] != SECTOR\_MAGIC)
* {
* j = 0;
* struct inode\_indirect\_block \*second\_block\_ind = (struct inode\_indirect\_block \*)malloc(BLOCK\_SECTOR\_SIZE);
* buffer\_cache\_read(second\_block\_ind->map\_table[j], second\_block\_ind, 0, sizeof(struct inode\_indirect\_block), 0);
* *while* (second\_block\_ind->map\_table[j] != SECTOR\_MAGIC)
* {
* free\_map\_release(second\_block\_ind->map\_table[j], 1);
* j++;
* }
* free\_map\_release(first\_block\_ind->map\_table[i], 1);
* free(second\_block\_ind);
* free\_done++;
* i++;
* }
* free(first\_block\_ind);
* free\_done = 0;
* *return*;
* }
* *if* (inode\_disk->indirect\_block\_sec != SECTOR\_MAGIC)
* {
* free\_done ++;
* struct inode\_indirect\_block \*block\_indirect = (struct inode\_indirect\_block \*)malloc(BLOCK\_SECTOR\_SIZE);
* buffer\_cache\_read(inode\_disk->indirect\_block\_sec, block\_indirect, 0, sizeof(struct inode\_indirect\_block), 0);
* *for*(int k=0; k<10; k++)
* free\_done++;
* i=0;
* *while* (block\_indirect->map\_table[i] != SECTOR\_MAGIC)
* {
* free\_map\_release(block\_indirect->map\_table[i], 1);
* i++;
* }
* free(block\_indirect);
* free\_done = 0;
* *return*;
* }
* i = 0;
* *for* (;inode\_disk->direct\_map\_table[i] != SECTOR\_MAGIC;)
* {
* free\_map\_release(inode\_disk->direct\_map\_table[i], 1);
* i++;
* }
* *return*;
* }
* Subdirectory
  + struct dir \*path\_parse(char \*name, char \*file\_name): path name을 절대/상대 경로로 구분하여 디렉토리 정보를 dir에 저장하고 path\_name의 디렉토리 정보와 파일 이름을 저장한다.
* struct dir \*path\_parse(char \*name, char \*file\_name)
* {
* int if\_parsed = 0;
* struct dir \*dir = NULL;
* *if*(!name || !file\_name || strlen(name) == 0)
* *return* NULL;
* char \*path = (char \*)malloc(sizeof(char) \* (PATH\_MAX\_LEN + 1));
* strlcpy(path, name, PATH\_MAX\_LEN);
* if\_parsed = 1;
* *if*(path[0] == '/')
* {
* *if*(if\_parsed == 1)
* if\_parsed = 0;
* dir = dir\_open\_root();
* }
* *else*
* {
* *if*(if\_parsed == 0)
* if\_parsed = 1;
* dir = dir\_reopen(thread\_current()->cur\_dir);
* }
* if\_parsed = 0;
* char \*ptr\_cur, \*ptr\_next, \*ptr\_tmp;
* ptr\_cur = strtok\_r(path, "/", &ptr\_tmp);
* ptr\_next = strtok\_r(NULL, "/", &ptr\_tmp);
* *if*(dir == NULL)
* *return* NULL;
* *if*(!inode\_is\_dir(dir\_get\_inode(dir)))
* *return* NULL;
* *while*(ptr\_cur != NULL && ptr\_next != NULL)
* {
* if\_parsed++;
* struct inode \*inode = NULL;
* *if* (!dir\_lookup(dir, ptr\_cur, &inode) || !inode\_is\_dir(inode))
* {
* if\_parsed--;
* dir\_close(dir);
* *return* NULL;
* }
* dir\_close(dir);
* dir = dir\_open(inode);
* ptr\_cur = ptr\_next;
* ptr\_next = strtok\_r(NULL, "/", &ptr\_tmp);
* }
* *if*(ptr\_cur == NULL)
* strlcpy(file\_name, ".", PATH\_MAX\_LEN);
* *else*
* strlcpy(file\_name, ptr\_cur, PATH\_MAX\_LEN);
* free(path);
* if\_parsed = 0;
* *return* dir;
* }
* */\** 
  + bool inode\_is\_dir(struct inode \*inode): inde\_disk 자료구조를 메모리에 할당하여 in-memory inode의 on-disk inode를 읽어 inode\_disk에 저장한다.
* bool inode\_is\_dir(struct inode \*inode)
* {
* bool result;
* *if* (inode->removed)
* {
* *return* false;
* }
* struct inode\_disk \*tmp\_disk\_inode = (struct inode\_disk \*)malloc(BLOCK\_SECTOR\_SIZE);
* buffer\_cache\_read(inode->sector, tmp\_disk\_inode, 0, BLOCK\_SECTOR\_SIZE, 0);
* result = tmp\_disk\_inode->is\_dir;
* free(tmp\_disk\_inode);
* *return* result;
* }
  + bool create\_dir(char \*name): bitmap에서 inode sector 번호를 할당 받고, 해당 sector에 file\_name의 디렉토리를 생성한다. 그리고 디렉토리 엔트리에 file\_name entry를 추가하고 ‘.’ 파일의 엔트리를 추가한다.
* bool create\_dir(char \*name)
* {
* int dir\_created = 0;
* block\_sector\_t inode\_sector = 0;
* char \*parsed\_name = (char \*)malloc(sizeof(char) \* (PATH\_MAX\_LEN+1));
* struct dir \*dir = path\_parse(name, parsed\_name);
* dir\_created = 1;
* bool success = (dir != NULL
* && free\_map\_allocate (1, &inode\_sector)
* && dir\_create (inode\_sector, 16)
* && dir\_add (dir, parsed\_name, inode\_sector));
* *for*(int i=0; i<5; i++)
* dir\_created ++;
* *if*(success)
* {
* struct dir \*tmp\_dir = dir\_open(inode\_open(inode\_sector));
* dir\_add(tmp\_dir, ".", inode\_sector);
* dir\_add(tmp\_dir, "..", inode\_get\_inumber(dir\_get\_inode(dir)));
* dir\_created++;
* dir\_close(tmp\_dir);
* dir\_created--;
* free(parsed\_name);
* dir\_close(dir);
* dir\_created = 1;
* *return* true;
* }
* *else*
* {
* dir\_created--;
* *if*(inode\_sector)
* {
* free\_map\_release(inode\_sector, 1);
* }
* free(parsed\_name);
* dir\_created = 0;
* dir\_close(dir);
* *return* false;
* }
* dir\_created = 0;
* *return* false;
* }
  + bool change\_dir(char \*path) (chdir syscall에서 사용): dir 경로를 분석하여 디렉토리를 반환한다. 그리고 스레드의 현재 작업 디렉토리를 변경한다.
* bool change\_dir(char \*path)
* {
* int dir\_changed = 0;
* char \*tmp = (char \*)malloc(sizeof(char) \* (PATH\_MAX\_LEN+1));
* strlcpy(tmp, path, PATH\_MAX\_LEN);
* strlcat(tmp, "/0", PATH\_MAX\_LEN);
* dir\_changed = 1;
* char \*parsed\_name = (char \*)malloc(sizeof(char) \* (PATH\_MAX\_LEN+1));
* struct dir \*dir = path\_parse(tmp, parsed\_name);
* *for*(int i=0; i<5; i++)
* dir\_changed++;
* *if*(dir == NULL)
* {
* dir\_changed = 0;
* free(tmp);
* free(parsed\_name);
* *return* false;
* }
* dir\_changed=0;
* free(tmp);
* free(parsed\_name);
* dir\_close(thread\_current()->cur\_dir);
* thread\_current()->cur\_dir=dir;
* *return* true;
* }
  + int readdir(int fd, char \*name): fd 리스트에서 fd에 대한 file 정보를 얻어오고, fd의 file->inode가 디렉토리인지 검사한다.
* int readdir(int fd, char \*name)
* {
* int flag = 0;
* bool result = true;
* struct file \*file = thread\_current()->fd[fd];
* struct inode \*inode = file\_get\_inode(file);
* *if*(inode == NULL)
* {
* *return* false;
* }
* *if*(!inode\_is\_dir(inode))
* {
* *return* false;
* }
* flag = 1;
* struct dir \*dir = dir\_open(inode);
* dir->pos = file\_tell(file);
* result = dir\_readdir(dir, name);
* file\_seek(file, dir->pos);
* flag = 0;
* *return* result;
* }
  + filesys\_remove (const char \*name): root 디렉터리에 파일 생성하는 대신 name 경로에 파일을 생성하도록 변경한다. 파일 생성 시, 절대/상대 경로를 분석해서 디렉토리 파일을 생성하도록 수정한다.
* bool
* filesys\_remove (const char \*name)
* {
* char \*parsed\_name = (char \*)malloc(sizeof(char)\*(PATH\_MAX\_LEN+1));
* char \*tmp = (char \*)malloc(sizeof(char)\*(PATH\_MAX\_LEN+1));
* struct dir \*dir = path\_parse(name, parsed\_name);
* bool success = false;
* struct inode \*inode;
* dir\_lookup(dir, parsed\_name, &inode);
* *if*(inode\_is\_dir(inode))
* {
* struct dir \*new = NULL;
* new = dir\_open(inode);
* *if*(new)
* {
* *if*(!dir\_readdir(new, tmp))
* {
* success = dir != NULL && dir\_remove (dir, parsed\_name);
* }
* dir\_close(new);
* }
* }
* *else*
* {
* success = dir != NULL && dir\_remove (dir, parsed\_name);
* }
* dir\_close (dir);
* free(parsed\_name);
* free(tmp);
* *return* success;
* }
* Buffer cache
  + buffer\_cache\_entry 구조체 및 buffer\_cache\_entry cache[NUM\_CACHE] 배열
* struct buffer\_cache\_entry {
* bool dirty;
* bool used;
* block\_sector\_t sector;
* bool clock\_bit;
* struct lock lock;
* void \*data;
* };
* static struct buffer\_cache\_entry cache[NUM\_CACHE];
  + void buffer\_cache\_init(void): buffer cache를 초기화하는 함수이다.
* void buffer\_cache\_init(void)
* {
* p\_buffer\_cache = (uint8\_t \*)malloc(sizeof(uint8\_t) \* NUM\_CACHE \* BLOCK\_SECTOR\_SIZE);
* *for*(int i=0; i<NUM\_CACHE; i++)
* {
* memset(&cache[i], 0, sizeof(struct buffer\_cache\_entry));
* cache[i].data = p\_buffer\_cache + (BLOCK\_SECTOR\_SIZE \* i);
* lock\_init(&(cache[i].lock));
* }
* clock\_hand = cache;
* lock\_init(&buffer\_cache\_lock);
* }
  + struct buffer\_cache\_entry \*buffer\_cache\_lookup(block\_sector\_t sector): buffer cache를 순회하며 target sector가 존재하는지 검색한다.
* struct buffer\_cache\_entry \*buffer\_cache\_lookup(block\_sector\_t sector)
* {
* struct buffer\_cache\_entry \*found = NULL;
* int lookup\_cnt = 0;
* lock\_acquire(&buffer\_cache\_lock);
* lookup\_cnt++;
* *for*(int i=0; i<NUM\_CACHE; i++) {
* lock\_acquire(&(cache[i].lock));
* *if*(cache[i].used && (cache[i].sector == sector))
* found = &cache[i];
* lock\_release(&(cache[i].lock));
* }
* *for*(int i=0; i<5; i++)
* lookup\_cnt ++;
* lock\_release(&buffer\_cache\_lock);
* lookup\_cnt = 0;
* *return* found;
* }
  + bool buffer\_cache\_read(block\_sector\_t sector\_idx, void \*buffer, off\_t bytes\_read, int chunk\_size, int sector\_ofs): buffer cache에서 요청 받은 buffer frame을 읽어온다. buffer\_cache\_write도 비슷한 형태로 구현한다.
* bool buffer\_cache\_read(block\_sector\_t sector\_idx, void \*buffer, off\_t bytes\_read, int chunk\_size, int sector\_ofs)
* {
* struct buffer\_cache\_entry \*lookup\_res;
* lookup\_res = buffer\_cache\_lookup(sector\_idx);
* int read\_cnt = 0;
* *if*(lookup\_res == NULL){
* lookup\_res = select\_victim();
* *if*(lookup\_res == NULL) *return* false;
* lock\_acquire(&lookup\_res->lock);
* lookup\_res->sector = sector\_idx;
* lookup\_res->used = true;
* block\_read(fs\_device, sector\_idx, lookup\_res->data);
* read\_cnt++;
* lock\_release(&lookup\_res->lock);
* }
* lock\_acquire(&lookup\_res->lock);
* memcpy(buffer + bytes\_read, lookup\_res->data + sector\_ofs, chunk\_size);
* *for*(int i=0; i<5; i++)
* read\_cnt ++;
* lookup\_res->clock\_bit = true;
* lock\_release(&lookup\_res->lock);
* read\_cnt = 0;
* *return* true;
* }
  + struct buffer\_cache\_entry \*select\_victim(void): Clock 알고리즘을 이용해 buffer cache에 자리가 없을 경우 buffer cache에서 victim을 선정하여 entry head 포인터를 반환한다.
* struct buffer\_cache\_entry \*select\_victim(void)
* {
* struct buffer\_cache\_entry \*victim = NULL;
* int victim\_flag = 0;
* lock\_acquire(&buffer\_cache\_lock);
* *for*( ; victim == NULL; )
* {
* lock\_acquire(&clock\_hand->lock);
* victim\_flag++;
* *if*((clock\_hand->used == 0) || (clock\_hand->clock\_bit == 0)){
* victim = clock\_hand;
* }
* clock\_hand->clock\_bit = false;
* lock\_release(&clock\_hand->lock);
* *if*(clock\_hand == cache + (NUM\_CACHE - 1))
* clock\_hand = cache;
* *else*
* clock\_hand++;
* }
* lock\_acquire(&victim->lock);
* victim\_flag--;
* *if*(victim->dirty == 1)
* buffer\_cache\_flush\_entry(victim);
* lock\_release(&victim->lock);
* lock\_release(&buffer\_cache\_lock);
* victim\_flag = 0;
* *return* victim;
  + void buffer\_cache\_flush\_entry(struct buffer\_cache\_entry \*b\_flush\_entry): 인자로 주어진 entry의 dirty bit를 false로 설정하고, 해당 내역을 disk로 flush한다.
* void buffer\_cache\_flush\_entry(struct buffer\_cache\_entry \*flush\_entry)
* {
* block\_write(fs\_device, flush\_entry->sector, flush\_entry->data);
* flush\_entry->dirty = false;
* }
  + void buffer\_cache\_terminate(void): 모든 dirty entry를 flush하고 buffer cache를 해지한다.
* void buffer\_cache\_terminate(void)
* {
* buffer\_cache\_flush\_all();
* free(p\_buffer\_cache);
* }
  + void buffer\_cache\_flush\_all(void): Buffer cache를 순회하며 dirty bit가 true인 entry를 모두 디스크로 flush한다.
* void buffer\_cache\_flush\_all(void)
* {
* lock\_acquire(&buffer\_cache\_lock);
* *for*(int i=0; i<NUM\_CACHE; i++)
* {
* lock\_acquire(&cache[i].lock);
* *if*(cache[i].dirty == 1)
* block\_write(fs\_device, cache[i].sector, cache[i].data);
* cache[i].dirty = false;
* lock\_release(&cache[i].lock);
* }
* lock\_release(&buffer\_cache\_lock);
* }
  1. **시험 및 평가 내용**