+----------------------------+

| CS 330 |

| PROJECT 4 : FILE SYSTEMS|

| DESIGN DOCUMENT |

+----------------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

박승호 <eric0514@kaist.ac.kr>

이태룡 <eoyoung2@kaist.ac.kr>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.

남은 토큰 전부 사용하겠습니다. (4개)

>> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjR_aTDifzrAhV5KqYKHaY7CqsQFjAAegQIBRAB&url=http%3A%2F%2Fesos.hanyang.ac.kr%2Ffiles%2Fcoursesware%2Fundergraduate%2FPINTOS%2FPintos_all.pdf&usg=AOvVaw3vxzSyKCzxVd9_HqrfpcyX

INDEXED AND EXTENSIBLE FILES

==========

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

```
struct inode_disk
  {
    //block_sector_t start;                 /* First data sector. */
    off_t length;                           /* File size in bytes. */
    unsigned magic;                         /* Magic number. */
    //uint32_t unused[125];                 /* Not used. */

    //modified 4.3
    uint32_t *is_dir;                   //file=0, director=1

    //modified 4.2
    block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES];
    block_sector_t indirect_block_sec;
    block_sector_t double_indirect_block_sec;
  };
```

블록의 위치를 direct, indirect, double indirect 방식으로 표현하도록 block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES], block_sector_t indirect_block_sec, block_sector_t double_indirect_block_sec, 3개의 변수를 삽입.

```
    struct lock extend_lock;
```

struct inode에 lock 변수 extend_lock을 추가. struct inode_disk data 변수를 제거.

```
enum direct_t
{
   NORMAL_DIRECT,
   INDIRECT,
   DOUBLE_INDIRECT,
   OUT_LIMIT
};
```

디스크 블록 번호를 가리키는 방식을 나타낸다. NORMAL_DIRECT는 inode에 직접 디스크 블록 번호를 저장한다. INDIRECT는 1개의 인덱스 블록을 사용하여 디스크 블록 번호에 접근함을 의미한다. DOUBLE_INDIRECT는 2개의 인덱스 블록을 사용하여 디스크 블록 번호에 접근함을 의미한다. OUT_LIMIT은 잘못된 파일의 오프셋일 경우를 의미한다.

```
struct sector_location
{
   int directness;
   int index1;
   int index2;
};
```

블록 주소의 접근 방식과 , 인덱스 블록내의 오프셋 값을 저장한다. directness는 접근 방식의 종류를 나타낸다. index1은 첫번째 인덱스 블록에서 접근할 엔트리의 오프셋을 나타낸다. index2는 두번째 인덱스 블록에서 접근할 값을 의미한다.

struct inode_indirect_block

{

  block_sector_t map_table[INDIRECT_BLOCK_ENTRIES];

};

인덱스 블록을 표현하는 자료구조이다.

>> A2: What is the maximum size of a file supported by your inode structure? Show your work.

    block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES];

여기에서 direct block이 124개임을 알 수 있다 (DIRECT_BLCOK_ENTRIES = 123). indirect_block과 double_indirect_block은 각각 한 개씩 존재 가능하다. direct_block의 크기는 1섹터, 즉 512bytes이고, 각 인덱스 블록은 128개 (BLOCK_SECTOR_SIZE / sizeof(block_sector_t)) 엔트리들을 갖는다.

512*124(direct) + 512*128*1(indirect) +512 * 128 * 128 * 1(double indirect) = 8380KB이다.

---- SYNCHRONIZATION ----

>> A3: Explain how your code avoids a race if two processes attempt to extend a file at the same time.

struct inode에 extend_lock 변수를 추가하고, 파일 변경 시 lock을 획득하여 사용하게 해, race 조건을 방지하였다.

>> A4: Suppose processes A and B both have file F open, both positioned at end-of-file. If A reads and B writes F at the same time, A may read all, part, or none of what B writes. However, A may not read data other than what B writes, e.g. if B writes non-zero data, A is not allowed to see all zeros. Explain how your code avoids this race.

파일을 쓰기 위한 파일 증가가 먼저 일어나 쓰기 프로세스가 lock에 의해서 보호가 될 것이다. 리드는 파일의 쓰기가 모두 일어나고 진행되어, race를 해결할 것이다.

>> A5: Explain how your synchronization design provides "fairness". File access is "fair" if readers cannot indefinitely block writers or vice versa. That is, many processes reading from a file cannot prevent forever another process from writing the file, and many processes writing to a file cannot prevent another process forever from reading the file.

inode에서 read는 락을 사용하지 않았기 때문에 읽는 도중에 쓰기 작업은 가능하다. 하지만, 쓰기 작업 도중에는 락을 사용하였기 때문에, 다소 fairness를 충족시키지 못했다고 생각한다.

---- RATIONALE ----

>> A6: Is your inode structure a multilevel index? If so, why did you choose this particular combination of direct, indirect, and doubly indirect blocks? If not, why did you choose an alternative inode structure, and what advantages and disadvantages does your structure have, compared to a multilevel index?

지금 사용하는 구조는 multilevel index이다. double indirect block까지 사용했는데, 이는 PINTOS에서 파일 크기를 8MB까지 요구했기 때문이다. indirect block과 direct block 두 가지로는 약 4MB까지 밖에 파일 크기를 키울 수 없다. 또한, triple indirect block은 파일크기가 약 4GB까지 커지는 경우에 사용하기 시작하므로 PINTOS에서는 필요하지 않다. 따라서, double indirect block까지 사용하는 방식을 사용하였다.

SUBDIRECTORIES

==========

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

```
uint32_t is_dir;                //file=0, director=1
```

struct indoe_disk에 추가한 파일과 디렉터리를 구분하기 위한 변수.

---- ALGORITHMS ----

>> B2: Describe your code for traversing a user-specified path. How do traversals of absolute and relative paths differ?

주어지는 디렉토리 경로는 "/"를 기준으로 토큰화 시켜서 경로를 구분한다. 이 때, 경로의 시작이 "/"이면 root 디렉토리를 경로의 시작점으로 삼는다. ".."을 경로가 주어지는 디렉토리 경로의 시작점이면, 현재 위치한 디렉토리의 부모 디렉토리를 경로의 시작점으로 한다. 마지막으로, "." 경우에는 현재 디렉토리를 시작점으로 경로를 이동한다.

---- SYNCHRONIZATION ----

>> B4: How do you prevent races on directory entries? For example, only one of two simultaneous attempts to remove a single file should succeed, as should only one of two simultaneous attempts to create a file with the same name, and so on.

Inode에 존재하는 락을 통해서 방지한다. 정해진 디렉토리에 할당된 inode의 lock이 하나의 프로세스 만이 디렉토리를 변경할 수 있게 할 것이다.

>> B5: Does your implementation allow a directory to be removed if it is open by a process or if it is in use as a process's current working directory? If so, what happens to that process's future file system operations? If not, how do you prevent it?

경로로 건내 받은 디렉토리가 정상적인 디렉토리라면 가능하다. 디렉토리가 제거된 다음에는 dir_close로 디렉토리에 대한 접근이 불가능해진다.

---- RATIONALE ----

>> B6: Explain why you chose to represent the current directory of a process the way you did.

현재 디렉토리를 계속 열려 있도록 각 쓰레드가 유지하고, 그 덕분에 디렉토리의 하위 디렉토리들이나 파일에 대한 접근이 쉬워지기 때문이다. 또한, 작업 중인 디렉토리에 대해 system call이 작동하게 만들기 쉽기 때문이다.

BUFFER CACHE

==========

---- DATA STRUCTURES ----

>> C1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.

```
struct buffer_head
{
    bool dirty_flag;                        //해당 entry가 dirty?
    bool valid_flag;                        //해당 entry의 사용여부
    block_sector_t sector_addr;             //해당 entry의 disk sector 주소
    bool clock_bit;                         //clock bit for clock algorithm
    struct lock lock;                       //lock 변수
    void *buffer;                           //buffer cache entry 가리키는 데이
터 포인터
};
```

Dirty_flag는 해당 엔트리가 변경되었는지 나타냄을 표시한다. Valid_flag는 해당 엔트리가 사용중인가를 나타낸다. Sector_addr은 엔트리의 캐시된 섹터 번호를 나타낸다. Clock_bit는 clock 알고리즘에서 사용하기 위해서 선언된 변수이다. Lock 변수는 이 락을 관리하기 위해서 선언하였다. Buffer는 데이터 버퍼를 가리킨다.

---- ALGORITHMS ----

>> C2: Describe how your cache replacement algorithm chooses a cache block to evict.

clock알고리즘을 사용하여 선택하였다. 버퍼캐시 리스트를 순회하면서, clock_bit이 true인 버퍼헤드를 false로 바꾸고, clock_bit이 false인 헤드를 찾는다. False인 헤드를 찾으면 반환하고, 없는 경우 iteration의 시작점을 반환하게 된다.

>> C3: Describe your implementation of write-behind.

bc_term과 bc_select_victim에서 구현하였다. bc_flush_entry에서 dirty_flag가 1인 블록들을 디스크로 flush하는 것을 구현하였는데, bc_term인 경우 모든 dirty_flag가 1인 블록들을 flush 하였고, bc_select_victim은 victim_entry의 dirty_flag가 1인경우 flush하였다.

>> C4: Describe your implementation of read-ahead.

구현하지 않음

---- SYNCHRONIZATION ----

>> C5: When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block?

데이터를 읽고 쓸 때는 버퍼 헤드의 락을 얻어 읽고 쓰기 때문에, 다른 프로세스가 블록을 쫓아낼 수 없다.

>> C6: During the eviction of a block from the cache, how are other processes prevented from attempting to access the block?

Eviction도 마찬가지로, lock을 할당하고 진행하기 때문에 다른 프로세스가 블록에 엑세스 할 수 없다.

---- RATIONALE ----

>> C7: Describe a file workload likely to benefit from buffer caching, and workloads likely to benefit

from read-ahead and write-behind.

메모리에 접근하는 시간이 디스크 입출력 시간보다 빠르므로, 디스크 블록을 메모리에 저장하여 데이터 읽기/쓰기 작업의 시간을 줄일 수 있다. Read-ahead는 큰 파일을 읽는 것과 같이 연속적인 읽기에 효과적이다. Write-behind의 경우는 빠른 속도로 데이터 쓰기를 필요로 할 때 효과적인 방법이다.

SURVEY QUESTIONS

==========

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems
>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students in future quarters?

>> Any other comments?