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

| CS 330 |

| PROJECT 4 : FILE SYSTEMS|

| DESIGN DOCUMENT |

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

---- GROUP ----

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

김건호(Kunho Kim) <kaist984@kaist.ac.kr>

김철환(Cheolhwan Kim) <maxwell98@kaist.ac.kr>

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

pintos/src/filesys에서 make check나 make grade를 진행할 시에 dir-vine 테스트 케이스에서 한번 끊기는 현상이 발생합니다. 혹시 이런 상황이 발생한다면 다시 make check나 make grade를 해주시면 감사하겠습니다.

>> 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.

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.

in inode.c

/\* Make inode\_disk size to BLOCK\_SECTOR\_SIZE \*/

#define DIRECT\_BLOCK\_ENTRIES 123

#define INDIRECT\_BLOCK\_ENTRIES(BLOCK\_SECTOR\_SIZE/

sizeof(block\_sector\_t))

enum sector\_type {DIRECT, INDIRECT, DOUBLE\_INDIRECT, ERROR};

/\* On-disk inode.

Must be exactly BLOCK\_SECTOR\_SIZE bytes long. \*/

struct inode\_disk

{

off\_t length; /\* File size in bytes. \*/

unsigned magic; /\* Magic number. \*/

uint32\_t is\_file; /\* If file type == FILE this

variable become 1.

If file type == DIRECTORY this variable become 0. \*/

block\_sector\_t direct\_block\_table[DIRECT\_BLOCK\_ENTRIES];

/\* Table that saves the direct sectors. \*/

block\_sector\_t indirect\_sector; /\* Indirect sector. \*/

block\_sector\_t double\_indirect\_sector;/\* Double indirect sector. \*/

};

struct block\_location

{

int type;

int index1;

int index2;

};

struct indirect\_block

{

block\_sector\_t indirect\_block\_table[INDIRECT\_BLOCK\_ENTRIES];

};

/\* In-memory inode. \*/

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 lock; /\* Lock for extensible file \*/

};

기존 inode 구조체의 inode\_disk를 담는 data를 삭제하고 lock을 추가했습니다.

inode의 실질적 정보를 담고있는 inode\_disk 역시 extensible file 기법을 위해 start라는 block\_sector를 제거하고 여러 block\_sector\_t를 담기 위한 direct array, indirect block sector, double indirect block sector를 추가했습니다.

>> A2: What is the maximum size of a file supported by your inode

>> structure? Show your work.

block 하나당 512 byte를 저장할 수 있습니다. inode\_disk 하나에 block의 sector 들을 총 123 + 128 + 128\*128 = 16635개 저장할 수 있으므로 충분한 filedisk의 크기만 갖춰진다면 총 8,517,120 byte를 저장할 수 있습니다.

---- SYNCHRONIZATION ----

>> A3: Explain how your code avoids a race if two processes attempt to

>> extend a file at the same time.

inode에 추가해준 lock을 이용하여 inode의 disk 정보를 수정할 때 발생하는 race condition을 해결합니다. inode\_disk의 정보를 변경하는 구간은 inode\_read(), inode\_write\_at(), inode\_close() 세 함수 뿐입니다. 여기서 inode\_disk의 정보를 수정하는 구간 사이에 inode의 lock을 acquire, release 함으로써 synch문제를 해결하였습니다.

>> 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

>> nonzero data, A is not allowed to see all zeros. Explain how your

>> code avoids this race.

A3에서의 답변과 마찬가지로 inode\_disk의 정보를 수정하는 모든 경우에 inode lock을 걸어주었습니다. file size를 확장해주는 과정 역시 inode\_disk의 정보를 수정하는 작업이기 때문에 inode lock을 얻은 후 수행하는 작업일 것입니다. 따라서 A4의 문제 역시 inode lock을 이용해서 해결 가능합니다.

>> 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.

A3, A4번 문항에서는 inode\_disk의 정보를 수정하는 모든 과정을 inode의 lock을 취득한 뒤에 실행하여 race conditions을 해결했다고 답했습니다. 하지만 그렇게 되면 A5처럼 한 process만 lock을 독점하는 상황이 발생할 수 있습니다.

저희는 이 문제를 해결하기 위해 inode\_read, write\_at 에서 lock을 좀 더 유동적으로 acquire, release 하도록 코드를 수정했습니다. read, write 공통적으로 while loop를 돌며 block\_size(block\_size로 나눠 떨어지는 chunk\_size 입니다.)만큼의 data를 읽거나 쓰는 작업을 진행하는데, 이때 inode\_disk의 정보를 건드리지 않는 구역에서 잠깐 lock을 풀어주고, inode\_disk를 사용할때 다시 얻어주는 작업을 반복하면서 여러 process가 한 file에 공정하게 접근할 수 있도록 코드를 수정했습니다.

---- 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?

n-level indirect block은 이론상 128^n block만큼의 data를 저장할 수 있습니다. 하지만 n-level block은 실제 file의 data가 저장된 block\_sector\_t를 찾아가기 위해 n번의 buffer\_cache\_read를 수행해야 합니다. 심지어 전부 다른 block\_sector\_t에 대한 read입니다. 즉, 기하급수적으로 증가하는 file 저장능력을 갖는 대신에 file을 읽기 위한 시간이 더욱 많이 소요된다는 장단점이 있습니다. 따라서 저희 code 는 filesystem disk의 전체 크기가 8MB라는 pintos 의 조건과 inode\_disk구조체 크기가 BLOCK\_SECTOR\_SIZE 512byte라는 조건을 만족시키기 위해 123size의 direct block, 128\*1size의 indirect block, 128\*128\*1 size의 double indirect block를 섞어서 구현했습니다.

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.

In filesys/inode.c

/\* On-disk inode.

Must be exactly BLOCK\_SECTOR\_SIZE bytes long. \*/

struct inode\_disk

struct inode\_disk

{

/\* File size in bytes. \*/

off\_t length;

/\* Magic number. \*/

unsigned magic;

/\* If file type == FILE is\_file become 1.

If file type == DIRECTORY is\_file become 0. \*/

uint32\_t is\_file;

/\* Table that saves the direct sectors. \*/

block\_sector\_t direct\_block\_table[DIRECT\_BLOCK\_ENTRIES];

/\* Indirect sector. \*/

block\_sector\_t indirect\_sector;

/\* Double indirect sector. \*/

block\_sector\_t double\_indirect\_sector;

};

In filesys/directory.h

/\* When we create the inode, this define type determines the is\_file of the inode. \*/

#define TYPE\_DIR 0

#define TYPE\_FILE 1

In threads/thread.h

struct thread

{

...

/\* Current working directory of thread. \*/

struct dir \*dir;

...

};

---- ALGORITHMS ----

>> B2: Describe your code for traversing a user-specified path. How

>> do traversals of absolute and relative paths differ?

일단 thread\_create()에서 thread\_current()->dir가 NULL이 아니라면, 즉 작업 디렉토리가 root\_dir이 아니라면 자식을 만들 때 해당 작업 디렉토리를 자식에게도 물려줍니다. system call에서 create(filesys\_create()) 이나 remove(filesys\_remove()), open(filesys\_open()), mkdir(filesys\_mkdir())이 호출되면, 인자로 받은 name을 복사해준 후, parse\_path()를 진행합니다.

parse\_path()에서는 인자로 받은 name의 첫 character가 “/”이라면 절대경로로 간주하고 dir\_open\_root()를 통해 root\_dir를 열고, “/”이 아니라면 상대경로로 간주하고 dir\_reopen(thread\_current()->dir)을 통해 현재 스레드의 작업 디렉토리를 열어줍니다. 열어준 디렉토리가 모종의 이유로 삭제되었거나, TYPE\_FILE이라면 NULL을 반환합니다. 이후 strtok\_r()함수를 이용하여 “/”로 인자로 받은 name을 끊어주면서 tokenization을 진행합니다. 첫 strtok\_r()을 실행했을 때, 그 반환값이 “.”이라면 file\_name에 “.”을 저장해주고, 열어준 작업 디렉토리를 반환합니다. 이후, while loop를 통해 strtok\_r()를 진행하면서 token(파일이나 디렉토리 이름)이 현재 열린 디렉토리에 존재하는지 dir\_lookup()을 통해 검사합니다. 존재한다면 그 token이 파일인지 디렉토리인지 inode\_is\_file()을 통해 검사합니다. 그 token이 디렉토리라면 현재 열린 디렉토리를 닫아준 후, 그 token에 해당하는 디렉토리를 열어주어 하위 디렉토리로 이동합니다. 이후 strtok\_r()을 통해 반복 작업을 수행합니다. while loop가 끝나면 인자로 받은 file\_name에 마지막 token을 복사해주고, 현재 열린 디렉토리를 반환합니다. 그 이후 각자의 system call에 해당하는 내용을 수행합니다.

---- 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.

directory entries를 관리하기 위해 directory가 소유하고 있는 inode의 정보를 inode\_read\_at을 이용해 불러옵니다. 이 과정에서 이미 extensible file 파트에서 완료한 inode의 정보 입/출력시 걸어주는 inode lock이 있어 race condition이 해결됩니다. race condtion 발생없이 inode의 정보 inode\_disk를 가져오고 이후 dir가 소유한 하위 directory의 정보를 갱신합니다.

>> 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?

디렉토리를 제거하기 위해서는 filesys\_remove() 함수를 실행합니다. filesys\_remove()를 실행하면 parse\_path()를 통해 인자로 들어온 name을 parsing해주고, 해당 name을 가진 파일 혹은 디렉토리의 상위 디렉토리를 반환합니다. 이후 디렉토리에서 최하위의 inode가 파일 혹은 디렉토리인지 확인하기 위해 dir\_lookup()을 진행하는데, 만약 최하위의 name에 해당하는 inode가 디렉토리라면, 현재 스레드의 작업 디렉토리와 비교해줍니다. 이 둘이 같다면, remove를 취소하고 filesys \_remove()는 false를 반환합니다. 이로써, 프로세스의 현재 작업 디렉토리가 지워지는 것을 방지합니다.

---- RATIONALE ----

>> B6: Explain why you chose to represent the current directory of a

>> process the way you did.

질문이 잘 해석되지 않아, subdirectory를 왜 지금과 같은의 방식으로 구현했는지에 대해 설명하겠습니다. subdirectory를 구현하면서 중요한 점은 기존에 root\_dir 밖에 없던 구조를 하위 디렉토리가 생성될 수 있도록 해야하고, 기존에는 root\_dir 밖에 없어서 절대 경로와 상대 경로가 같았다면 subdirectory가 구현된 시점에서는 절대 경로와 상대 경로가 철저하게 구분되어야 합니다. 그러므로 경로를 참조해야하는 모든 함수들 (filesys\_create(), filesys\_remove(), filesys\_open(), filesys\_mkdir())에서 parse\_path를 통해 절대 경로와 상대 경로에 따라 열어주는 디렉토리를 다르게 했습니다. 상대 경로는 기본적으로 thread\_create()에서 root\_dir로 설정해주고, chdir()을 통해서 변경되었다면 해당 스레드 구조체의 dir 변수에 저장할 수 있도록 했습니다.

다른 한 가지 중요한 점은 파일이 생성되는지 디렉토리가 생성되는지를 inode가 알아야 한다고 생각을 했습니다. 그러므로 inode 구조체에 is\_file이라는 변수를 추가하여 해당 inode가 파일인지 디렉토리인지를 구분할 수 있게 하였습니다. 이에 따라, 기존의 filesys\_create()를 변형하여 파일만을 만들게끔 하였고, filesys\_mkdir()를 만들어주어 디렉토리만을 만들게끔 하였습니다.

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.

in buffer\_cache.h

struct buffer\_cache

{

struct inode \*inode; /\* Inode. \*/

block\_sector\_t sector; /\* Point sector if filedisk. \*/

bool using; /\* Whether this buffer cache is

using or not. \*/

bool dirty; /\* Check bc is dirty. \*/

bool chance; /\* Using at clock algorithm. If

chance is 1, bc can survive once. \*/

void \*data; /\* Buffer that saved real data.

\*/

struct lock lock; /\* Acquire this when accessing

buffer cache. \*/

};

in buffer\_cache.c

#define BUFFER\_CACHE\_ENTRIES 64

int clock\_hand = 0; /\* Using in clock algorithm \*/

static struct buffer\_cache buffer\_cache\_table[BUFFER\_CACHE\_ENTRIES]; /\* Manage buffer cache \*/

static char p\_buffer\_cache[BUFFER\_CACHE\_ENTRIES \* BLOCK\_SECTOR\_SIZE]; /\* Buffer that save real data \*/

struct lock bct\_lock;

---- ALGORITHMS ----

>> C2: Describe how your cache replacement algorithm chooses a cache

>> block to evict.

저희는 clock hand를 이용한 clock 알고리즘을 사용하여 방출할 buffer cache block을 선정하였습니다. 먼저 전역변수로 시계침 clock\_hand를 선언하고 buffer\_cache\_read 나 write를 진행할 때 lookup을 통해 같은 block sector를 보유한 cache block을 찾아내지 못한다면 victim cache block을 찾아내는 buffer\_cache\_victim()을 발동합니다. 이 함수는 전역변수로 선언된 clock\_hand를 1씩 더해주며 cache\_block의 table을 순차적으로 검사합니다. 이때 검사받는 block에서 시침이 한바퀴를 돌아오기 전에 한번 이상의 buffer\_cache\_read, write이 수행됐다면, block의 chance는 1이고 따라서 한번의 기회를 얻게 됩니다. (evict되지 않고 시침이 다음 block을 검사합니다.)

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

총 3가지 경우가 있습니다.

1. 기존 캐시에 있던 block이 evict되는 경우
2. clock\_hand 검사할 때 block이 chance를 갖고 있던 경우
3. 프로그램이 종료되는 과정에서 filesys\_done()이 호출 되어 block table의 모든 block을 flush할 때

세 가지 경우 모두 공통점이 있는데, 그건 buffer\_cache\_write가 한번이라도 진행되어 buffer\_cache의 dirty가 true인 경우에만 block\_write를 진행한다는 점입니다. 이걸로 불필요한 block\_write를 줄일 수 있었습니다.

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

buffer\_cache\_lookup을 이용하여 읽고 싶은 block\_sector\_t의 block이 이미 buffer\_cache로 등록되어 있는지 확인합니다. 등록되어 있다면, 불필요하게 block\_read를 할 필요없이 kernel영역에 있는 해당 buffer에 바로 read를 진행합니다. 이것으로 불필요한 block\_read를 줄일 수 있습니다. 그러므로 sequential한 data를 read하는 경우, 시간을 단축할 수 있습니다.

---- 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?

buffer\_cache의 구조체 안에 추가해놓은 lock을 이용해서 read, write시 lock을 acquire, release 함으로써 같은 buffer\_cache block에 동시에 접근하는 것을 차단하였습니다.

>> C6: During the eviction of a block from the cache, how are other

>> processes prevented from attempting to access the block?

buffer\_cache block 개별의 lock 말고도 buffer\_cache\_table을 관리하는 전체의 lock bct\_lock을 만들었습니다. table 전체를 순회하거나 clock\_hand를 이용해 table의 특정 buffer\_cache block을 참조할 때 항상 bct\_lock을 걸어두어 synch 문제를 해결했습니다.

---- RATIONALE ----

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

>> and workloads likely to benefit from read-ahead and write-behind.

read\_ahead 기법이 적용되어 이미 buffer\_cache table에 읽고자 하는 block\_sector의 정보가 담겨있다면, 속도가 느린 block\_read를 사용하지 않고 단지 kernel의 주솟값을 읽는 것만으로도 특정 block의 정보를 얻을 수 있습니다.

마찬가지로 write\_behind 기법을 적용함으로써 속도가 느린 block\_write의 빈도 수를 줄였습니다. (C3에서 줄어드는 경우를 기술했습니다.)

결론적으로 속도가 느린 block\_read, write를 획기적으로 줄임으로써 실행 속도를 향상시킬 수 있었습니다.

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?

extensible file을 구현하고나서 특정 테스트 케이스를 실행했을 때, kernel panic(ex. assertion ‘is\_thread (t)’)이 발생했고, 어디서 에러가 발생했는지 찾기 힘들었습니다.

특히 여러 file을 거쳐서 코드가 진행됐기 때문에 printf를 찍어서 확인할 때도 상당히 복잡했습니다.

평일에는 매일 12시부터 3~4시까지, 주말과 연휴에는 거의 하루종일 zoom 화상회의를 통해 직접 얘기를 주고받으며 아이디어 design과 코딩을 했습니다. 시험기간이라 아무래도 상대적으로 심리적 압박감이 심했고, OS 시험 바로 직전인 13일 일요일에도 특정 테스트 케이스를 통과하지 못해 고통받다가 결국 몇 가지 테스트 케이스를 해결하지 못하고 제출했습니다.

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

buffer cache와 read-ahead, write behind에 대한 것을 수업을 통해 먼저 접해서 실제로는 어떤 방식으로 작동할지 굉장히 궁금했는데, pintos를 구현하면서 궁금증이 많이 해소된 것 같습니다. 또 초반에는 extensible 파일에서 indirect block과 double indirect block에 대한 개념이 잘 잡히지 않아서 힘들었는데, pintos 설계를 디자인하면서 개념을 잡아갈 수 있었고, 그것을 구현함으로써 완전히 체득할 수 있었습니다. 마지막으로 subdirectory를 구현하면서 실제로 리눅스에서 사용되는 cd, mkdir, rm 등이 어떻게 작동되는지를 이해할 수 있었고, 생각외로 edge case가 많아서 잘 고려해주어야 한다는 점을 깨달았습니다. 이러한 edge case들은 직접 리눅스 shell에서 command를 이것저것(ex cd /../../, cd ~/pintos//src 등등) 시도해보면서 더 단단히 다져나갔습니다.

>> 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?

없습니다. 조교님들이 물심양면으로 도와주시니 항상 감사할 따름입니다.

>> Do you have any suggestions for the TAs to more effectively assist

>> students in future quarters?

딱히 없습니다.

>> Any other comments?

한 학기 동안 고생 많으셨습니다!!