**Pintos Project 5: Filesystem**

담당 교수 : 김영재

조 / 조원 :20171628 김현우

개발 기간 : 2021/12/1 ~ 2021/12/4

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

파일 시스템을 개선함으로써 파일 증가도 구현하고, 디렉토리를 루트 말고도 하위 디렉토리를 읽을 수도 있다. 추가적으로 버퍼캐시도 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Extensible file & file growth
2. Subdirectory
3. Buffer cache  
   **구현한 경우만 작성**

메모리에 임시공간을 만들어서 직접 디스크에 접근해서 패이지를 가져오는 대신 buffer\_cache 에 접근해서 페이지를 가져온다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Extensible file & file growth
  + Index structure와 management에 대해서 기술
* Subdirectory
  + Directory entry 관리 방법
* Buffer cache
  + Buffer cache eviction 방식

구현이 쉬운 clock algorithm 방식으로 구현하였다.

* + Buffer cache flush 방식

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

Filesystem 다이렉토리에 buffer\_cache.c 와 헤더파일을 추가한다. 그 안에 다음과 같은 함수를 추가한다.

struct buffer\_cache\_entry

{

    bool dirty\_bit;

    bool valid\_bit;

    bool reference\_bit;

    block\_sector\_t disk\_sector;

    struct lock lock;

    uint8\_t buffer[BLOCK\_SECTOR\_SIZE];

};

void bc\_init(void);

void bc\_term(void);

bool bc\_read(block\_sector\_t, void \*, off\_t, int, int);

bool bc\_write(block\_sector\_t, void \*, off\_t, int, int);

struct buffer\_cache\_entry \*bc\_lookup(block\_sector\_t);

struct buffer\_cache\_entry \*bc\_select\_victim(void);

void bc\_flush\_entry(struct buffer\_cache\_entry \*);

void bc\_flush\_all\_entries(void);

bc\_init()

버퍼캐시를 초기화 한다.

Bc\_term()

버퍼 캐시를 없엔다.

Bc\_read()

새로 BUFFER에 넣을 데이터를 읽어온다.

Bc\_write()

버퍼캐시에 있는 정보를 다른 블록으로 복사해준다.

Bc\_lookup()

전달받은 sector 값과 동일한 buffer cache entry 가 있는지 여부를 확인한다.

Bc\_select\_victim()

Evict할 buffer cache를 선택해서 buffer cache 에서 evict한다. Clock algorithm 활용함

Bc\_flush\_entry

전달받은 entry를 비우는 함수

Bc\_flush\_all\_entries

모든 entries를 비우는 함수

* 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
* Subdirectory
* Buffer cache  
  **구현한 경우만 작성**

void bc\_init(void)

{

    //lock\_init(&buffer\_cache\_lock);

    for (int i = 0; i != BUFFER\_CACHE\_ENTRIES; i++)

    {

        memset(&buffer\_head[i], 0, sizeof(struct buffer\_cache\_entry));

        lock\_init(&(&buffer\_head[i])->lock);

    }

    clock\_hand = buffer\_head;

}

Bc init함수에서는 buffer\_cache 를 순회하면서 entries 마다 0으로 초기화를 시켜준다. 또한 각각의 entreis 마다 lock을 활성화 한다.

void bc\_term(void)

{

    bc\_flush\_all\_entries();

}

Buffer cache를 종료하면서 flush\_all\_entries 를 부른다.

bool bc\_read(block\_sector\_t sector\_idx, void \*buffer, off\_t bytes\_read, int chunk\_size, int sector\_ofs)

{

    /\* sector\_idx를 buffer\_head에서 검색 (bc\_lookup 함수 이용) \*/

    /\* 검색 결과가 없을 경우, 디스크 블록을 캐싱 할 buffer entry의 buffer\_head를 구함 (bc\_select\_victim 함수 이용) \*/

    /\* block\_read 함수를 이용해, 디스크 블록 데이터를 buffer cache 로 read \*/

    /\* memcpy 함수를 통해, buffer에 디스크 블록 데이터를 복사 \*/

    /\* buffer\_head의 clock bit을 setting \*/

    struct buffer\_cache\_entry \*cur = bc\_lookup(sector\_idx);

    if (cur == NULL)

    {

        cur = bc\_select\_victim();

        bc\_flush\_entry(cur);

        cur->valid\_bit = true;

        cur->dirty\_bit = false;

        cur->disk\_sector = sector\_idx;

        block\_read(fs\_device, sector\_idx, cur->buffer);

    }

    cur->reference\_bit = true;

    memcpy(buffer + bytes\_read, cur->buffer + sector\_ofs, chunk\_size);

    lock\_release(&cur->lock);

    return true;

}

인자로 들어온 sector idx를 buffer\_cache에서 lookup한다. 만약에 존재 하지 않는다면 인자로 들어온 sector\_idx를 복사해서 buffer\_cache에 넣어준다. 만약에 해당 원소를 buffer\_cache에서 찾지 못한경우 bc\_select\_vitctim을 소환해서 evict할 원소를 고른다. 이때 valid bit을 true / dirty bit을 false 마지막으로 reference bit를 true로 처리해준다. 앞에서 걸어주었던 lock을 release를 해준다.

bool bc\_write(block\_sector\_t sector\_idx, void \*buffer, off\_t offset, int chunk\_size, int sector\_ofs)

{

    /\* sector\_idx를 buffer\_head에서 검색하여 buffer에 복사(구현)\*/

    /\* update buffer\_head (구현) \*/

    struct buffer\_cache\_entry \*cur = bc\_lookup(sector\_idx);

    if (cur == NULL)

    {

        cur = bc\_select\_victim();

        bc\_flush\_entry(cur);

        cur->valid\_bit = true;

        cur->disk\_sector = sector\_idx;

        block\_read(fs\_device, sector\_idx, cur->buffer);

    }

    cur->reference\_bit = true;

    cur->dirty\_bit = true;

    memcpy(cur->buffer + sector\_ofs, buffer + offset, chunk\_size);

    lock\_release(&cur->lock);

    return true;

}

인자로 들어온 sector id를 buffer cache에서 검색해서 buffer에 복사해준다. 만약에 없다면 들어온 인자를 buffer cache에 쓰고 다시 buffer에 복사해준다. 만약에 해당 원소를 그리고 bc\_select\_vitctim을 소환해서 evict할 원소를 고른다. 마지막으로 앞에서 걸어주었던 lock을 release를 해준다.

struct buffer\_cache\_entry \*bc\_lookup(block\_sector\_t sector)

{

    /\* buffe\_head를 순회하며, 전달받은 sector 값과 동일한 sector 값을 갖는 buffer cache entry가 있는지 확인 \*/

    /\* 성공 : 찾은 buffer\_head 반환, 실패 : NULL \*/

    for (int i = 0; i < BUFFER\_CACHE\_ENTRIES; i++)

    {

        if (buffer\_head[i].valid\_bit && buffer\_head[i].disk\_sector == sector)

        {

            lock\_acquire(&(&buffer\_head[i])->lock);

            return &buffer\_head[i];

        }

    }

    return NULL;

}

Buffer\_cache를 순회하면서 해당 원소가 buffer\_cache안에 있는지 확인한다. 이때 해당 원소를 찾았다면 해당 원소를 디른 스레드에서 접근하지 못하게 lock을 걸면서 return한다. 없다면 null을 리턴

struct buffer\_cache\_entry \*bc\_select\_victim(void)

{

    /\* clock 알고리즘을 사용하여 victim entry를 선택 \*/

    /\* buffer\_head 전역변수를 순회하며 clock\_bit 변수를 검사 \*/

    /\* 선택된 victim entry가 dirty일 경우, 디스크로 flush \*/

    /\* victim entry에 해당하는 buffer\_head 값 update \*/

    /\* victim entry를 return \*/

    int count = 0;

    int idx = (clock\_hand - buffer\_head) / sizeof(struct buffer\_cache\_entry);

    if (idx == BUFFER\_CACHE\_ENTRIES)

        idx = 0;

    for (; idx < BUFFER\_CACHE\_ENTRIES; idx++)

    {

        lock\_acquire(&(buffer\_head[idx].lock));

        if (!buffer\_head[idx].valid\_bit)

        {

            clock\_hand = &buffer\_head[idx + 1];

            return &buffer\_head[idx];

        }

        if (!buffer\_head[idx].reference\_bit)

        {

            clock\_hand = &buffer\_head[idx + 1];

            return &buffer\_head[idx];

        }

        buffer\_head[idx].reference\_bit = false;

        lock\_release(&(buffer\_head[idx].lock));

        if (idx == BUFFER\_CACHE\_ENTRIES - 1)

            idx = 0;

    }

}

수업시간에 배운 clock 알고리즘을 활용해서 reference bit이 1이면 0으로 만들고 만약에 reference bit 이 0이면 evitct한다. 단 valid bit 이 0이면 쓸모없는 페이지 이므로 당연히 evict를 해야한다.

void bc\_flush\_entry(struct buffer\_cache\_entry \*p\_flush\_entry)

{

    if (!p\_flush\_entry->valid\_bit)

        return;

    if (!p\_flush\_entry->dirty\_bit)

        return;

    p\_flush\_entry->dirty\_bit = false;

    block\_write(fs\_device, p\_flush\_entry->disk\_sector, p\_flush\_entry->buffer);

}

포인터로 들어온 entry를 buffer cache에서 제외시킨다.

void bc\_flush\_all\_entries(void)

{

    for (int i = 0; i < BUFFER\_CACHE\_ENTRIES; i++)

    {

        lock\_acquire(&(buffer\_head[i].lock));

        bc\_flush\_entry(&buffer\_head[i]);

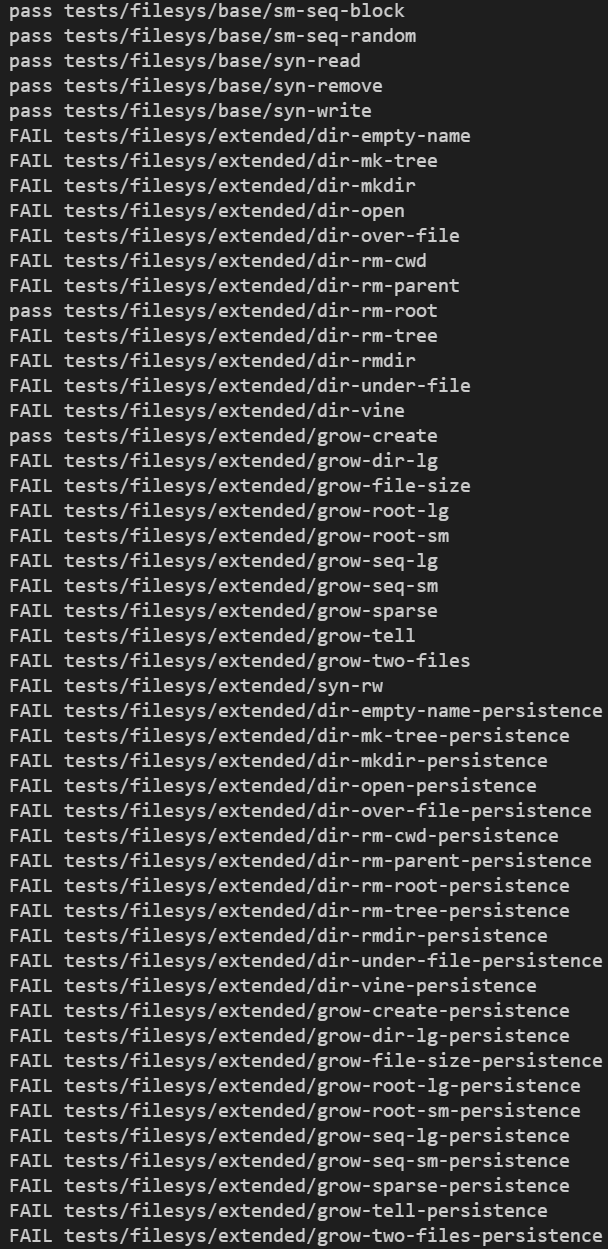
        lock\_release(&(buffer\_head[i].lock));

    }

}

모든 entries 를 flush하면서 buffer cache를 초기화 시킨다. 각각의 테이블을 flush 할때마다 해당 entries 의 lock을 얻어서 다른 스레드가 접근하지 못하게 한다.

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



**Buffer cache 만 구현해서 extened 구조는 돌아가지 않는 모습을 보이나, 이전의 파일 테스트 케이스는 잘 돌아가는 모습을 보여준다.**