CMOS 시간 출력 및 Buffer Cache 구현 보고서

과목명.	임베디드운영체제
참여 프로젝트.	앱 응답성 향상을 위한 선반입기
제출일.	2023년 12월 19일
7조	
2020061967	손윤석
2020008368	문예인
2020049752	유아영
2020044002	조정현

목차

1.1 프로젝트 소개

1.1.1 CMOS(Real-Time Clock) 시간 출력

CMOS와 Real Time Clock (RTC)는 시스템 설정 및 시간 정보를 저장하는데 사용되는 기술이다. 저전력 소모 특성을 가지고 있어 시스템이 꺼져 있어도 작동하며, 정확한 시간 데이터를 제공한다.

이 프로젝트는 CMOS와 RTC을 사용하여 현재 시간을 표시하는 것이다. CMOS와 RTC는 동일한 칩에 구현되어 있으며, RTC에서 시간 정보를 읽어와 사용자가 이해할 수 있는 형식으로 변환하여 출력하는 것이다.

1.2.1 Buffer Cache Simulator

해당 프로젝트는 파일 시스템에서의 데이터 액세스 효율성을 향상시키기 위한 "Buffer Cache Simulator"를 구현한다. 파일 시스템의 성능에 있어서 데이터 액세스는 핵심적인 요소이며 특히, 디스크에서 데이터를 읽어오는 과정에서 빈번한 액세스는 시스템 성능에 큰 영향을 미친다. 이 프로젝트는 Buffer Cache의 역할을 분석하고 구현하여, 실제 파일 시스템에서의 동작을 모방하고 최적화하는 데 중점을 둔다.

1.2 프로젝트 달성 목표

1.2.1 CMOS(Real-Time Clock) 시간 출력 프로젝트 목표

1.2.1.1 CMOS RTC에서 읽어온 시간 데이터를 사용자가 쉽게 이해할 수 있는 형식으로 변환하고 일반적인 시계형식으로 변환하여 ('YYYY-MM-DD HH:MM:SS') 사용자 인터페이스에 출력한다.

1.2.1.2 CMOS에서 RTC(Real-Time Clock)을 통해 제공되는 시간 데이터의 정확성을 인터넷 기반 서버와 같은 다른 시간 소스와 비교하여 정확성을 검증한다.

1.2.2 Buffer Cache Simulation 구현 프로젝트 목표

1.2.2.1 Buffer read hit과 miss의 경우를 명확히 구분하고, 소요되는 시간을 고려하여 Buffering을 위한 자료 구조 및 탐색 구조를 효율적으로 설계한다.

1.2.2.2 비어 있는 Buffer entry에 대한 Buffered Write을 구현하고, Dirty Buffer entry를 Flush Thread를 통해 디스크에 비동기적으로 기록하고 Flush 중인 Buffer entry에 대한 새로운 Write 요청이 있을 경우, 동기화 메커니즘을 통해 처리한다.

이를 통해 Buffered Write가 빠르게 수행되고, Dirty Buffer entry가 비동기적으로 디스크에 쓰여지는 것을 목표로 한다.

1.2.2.3 Buffer Cache의 세 가지 Replacement Policy 알고리즘을 구현한다. 1) FIFO 2) LRU 3) LFU

1.3 추진 배경 및 필요성

1.3.1 I/O Programming 실습을 통한 기초 지식 습득

CMOS와 같은 실제 하드웨어 컴포넌트를 조작함으로써, 레지스터와 메모리에 접근하는 기본적인 I/O 명령어들의 사용법을 익히게 된다.

1.3.2 Buffer Cache의 이해 및 실습

Buffer Cache Simulation을 구현하면서 시스템 아키텍처에 대한 이해를 직관적으로 할 수 있게 되며 데이터 액세스의 성능과 효율성을 향상시키기 위해서 시스템에서 어떻게 동작하는지 실질적인 이해를 얻게 된다. 또한 FIFO, LRU 그리고 LFU등의 Replacement Policy 알고리즘을 학습하고 어떤 알고리즘이 효율적인지 실제 시나리오를 통해 경험 할 수 있게 된다.

1.4 예상 결과물

1.4.1 CMOS(Real-Time Clock) 시간 출력

CMOS를 통해 실시간 시간 정보를 읽어와 사용자에게 표시하는 프로그램. 사용자가 이해하기 쉬운 형태로 시간을 출력합니다

1.4.2 Buffer Cache Simulator 개발

Application ⟷ Library ⟷ Operating System ⟷ Device의 구조를 갖는 Buffer Cache Simulator를 개발한다

Buffered Read, Delayed Write 그리고 Replacement Policy 3가지(FIFO, LRU, LFU)를 포함하여 구현.

2.1 업무분장

문예인: Buffer Cache 검증 및 테스트/중간, 결과 보고서 작성

손윤석: Buffer cache 설계 및 구현/중간, 결과 보고서 작성

유아영: Replacement Policy 구현/중간, 결과 보고서 작성

조정현: CMOS 출력/중간, 결과 보고서 작성

2.2 프로젝트 수행 일정

11월 20일 ~ 11월 25일

- 배경지식 습득/자료조사/Replacement Policy 선정
- 수행계획서 작성

11월 26일 ~ 11월 30일

- CMOS 시간출력 구현
- 중간보고서 작성

12월 1일 ~ 12월 7일

- Buffer Cache Simulator 뼈대코드 이해 및 설계

12월 8일 ~ 12월 14일

- Buffer Cache Simulator 구현

12월 15일 ~ 12월 19일

- Buffer Cache Simulator 검증 및 테스트
- 결과보고서 작성

3.1 주요기능 및 상세 설명

3.1.1 CMOS(Real-Time Clock) 시간 출력

CMOS의 메모리 어드레스를 참조한다.

```
#define year_register  0x09
#define month_register  0x08
#define day_register  0x07
#define hour_register  0x04
#define minute_register  0x02
#define second_register  0x00
```

해당 BCD 포멧을 bcd_to_decimal 함수를 사용하여 Binary로 변환한다.

```
int bcd_to_decimal(unsigned char bcd) {
    return ((bcd >> 4) * 10) + (bcd & 0x0F);
}
```

read_cmos 함수를 사용하여 RTC의 레지스터에서 날짜 및 시간 정보를 읽어온 후 출력문을 작성한다.

Date: year / month / day Time: hour : minute : second 의 형태로 출력한다.

```
int year = bcd_to_decimal(read_cmos(year_register));
int month = bcd_to_decimal(read_cmos(month_register));
int day = bcd_to_decimal(read_cmos(day_register));
int hour = bcd_to_decimal(read_cmos(hour_register));
int minute = bcd_to_decimal(read_cmos(minute_register));
int second = bcd_to_decimal(read_cmos(second_register));

//print utc time
int utc_hour = hour;
printf("CMOS_RTC\nDate:%04d/%02d/%02d Time:%02d:%02d:%02d UTC\n", 2000 + year, month, day, utc_hour, minute, second);
```

CMOS 값을 참조하면 UCT 값을 가져오기 때문에 KST 도 함께 출력하게 하였다.

```
    //print utc time
    int utc_hour = hour;
    printf("CMOS_RTC₩nDate:%04d/%02d/%02d Time:%02d:%02d:%02d UTC₩n", 2000 + year, month, day, utc_hour, minute, second);
    //KST = UTC+9
    hour += 9;
    // overflow in hour
    if (hour >= 24) {
        hour -= 24;
        day += 1;

        int days_in_month[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

        if (is_leap_year(2000 + year)) {
            days_in_month[2] = 29;
        }

        if (day > days_in_month[month]) {
            day = 1;
            month += 1;

            if (month > 12) {
                month = 1;
                year += 1;
            }
        }
    }
    //print kst
    printf("Date:%04d/%02d/%02d Time:%02d:%02d:%02d KST₩n", 2000 + year, month, day, hour, minute, second);
    return 0;
}
```

UTC ➔ KST

해당 프로그램을 sudo 권한 없이 실행시에는 오류를 출력한다.

```
[12/16/23]seed@VM:~/hy$ ./cmos
ret: -1)
```

sudo 권한을 부여하고 실행 시 정상적으로 레지스터에서 읽어온 시간 값을 출력하는 것을 확인할 수 있다.

```
[12/16/23]seed@VM:~/hy$ sudo ./cmos
CMOS_RTC
Date:2023/12/16 Time:05:07:21 UTC
Date:2023/12/16 Time:14:07:21 KST
```

3.1.2 Buffer Cache Simulator

3.1.2.1 Buffer cache 구조 설명 및 주요 함수 설명

버퍼 캐시는 queue.h, queue.c, stack.h, stack.c, buffer.c, cachebuffer.c, cachebuffer.h로 구성되어 있고 버퍼 캐시의 검증과 테스트는 test.c에서 이루어진다.

```
struct block {
    int block_nr;
    char data[BLOCK_SIZE];
    int dirty_bit; // 1 disk에 쓰이지 않음 0 쓰임
    int ref_count; // 참조횟수
};


typedef struct node {
    struct block *blk;
    struct node *next;
} Node;

typedef struct buffercache {
    Node *array[CACHE_SIZE];
    int items; // node의 개수 총합
    Queue *cachequeue; // FIFO에서 사용됨
    Stack *cachstack; // LRU에서 사용됨
} BufferCache;
```

Buffer Cache는 **hash table** 구조를 사용하여 정의했다.

Replacement Policy 인 **FIFO**를 위해 **Queue** 구조인 **cachequeue**,

LRU를 위해 **Stack** 구조인 **cachestack**, **LFU**를 위해 변수 **ref_count** 선언했다.

1. Buffered Read

A. os_read 함수

```c
int os_read(int block_nr, char *user_buffer)
{
    int ret;

    // implement BUFFERED_READ
    if (buffered_read(bc, block_nr, user_buffer) == 0) {
        return 0;
    }

    ret = lseek(disk_fd, block_nr * BLOCK_SIZE, SEEK_SET);
    if (ret < 0)
        return ret;

    ret = read(disk_fd, disk_buffer, BLOCK_SIZE);
    if (ret < 0)
        return ret;

    memcpy(user_buffer, disk_buffer, BLOCK_SIZE);
    fprintf(stdout, "%.10s...\n\n", user_buffer);

    return ret;
}
```

buffered_read를 호출하여 블록이 버퍼 캐시에 있는지 확인한다.

B. buffered_read 함수

```c
int buffered_read(BufferCache *buffercache, int block_nr, char *result)
{
    // 시간 측정
    clock_t start, end;
    start = clock();

    if (block_nr < 0 || block_nr > DISK_BLOCKS - 1)
    {
        return -1;
    }
    int index = hash(block_nr);
    Node *current = buffercache->array[index];
    while (current != NULL)
    {
        // found
        if (current->blk->block_nr == block_nr)
        {
            memcpy(result, current->blk->data, BLOCK_SIZE);
            current->blk->ref_count++;

            // 스택의 맨 위로 push
            push(buffercache->cachstack, block_nr);

            end = clock();
            fprintf(stdout, "[ Hit ]\t\tBlock numder: %d, data: \"%.10s...\" time: %d ms\n", block_nr, result, (int)(end - start));
            return 0;
        }
        current = current->next;
    }
    end = clock();

    fprintf(stdout, "[ Miss ]\tBlock numder: %d, time: %d ms\n", block_nr,(int)(end - start));
    // direct_IO
    char *test_buf = (char *)malloc(BLOCK_SIZE);
    // int disk_fd = open("diskfile", O_RDWR|O_DIRECT);
    if (lseek(disk_fd , block_nr  * BLOCK_SIZE, SEEK_SET) < 0)
        perror("lseek");
    if (read(disk_fd, disk_buffer, BLOCK_SIZE) < 0)
        perror("read");
    pthread_mutex_unlock(&lock);
    return -1;
}
```

1) 주어진 블록 번호를 이용하여 hash 함수를 통해 해당 블록이 버퍼 캐시 어느 인덱스에 위치하는지 계산한다.

   i. 만약 버퍼 캐시에 있다면 Hit로 처리하고 데이터를 사용자 버퍼로 복사한다.

      - 연결 리스트를 순회하면서 블록을 찾고, 블록의 데이터를 result 버퍼에 복사.
      - 찾은 블록의 ref_count를 증가[LFU]
      - 찾은 블록의 번호를 캐시 스택에서 제거 후 스택의 맨 위로 push[LRU]

   ii. Miss인 경우, 디스크에서 읽어와 버퍼 캐시를 업데이트하는 direct I/O를 시도

2) Hit와 Miss를 기록하며, 소요된 시간을 포함한 로그를 출력

2. Delayed Write

```c
int delayed_write(BufferCache *buffercache, int block_nr, char *input, int mode)
{
    if (block_nr < 0 || block_nr > DISK_BLOCKS - 1)
    {
        return -1;
    }

    int ret = 0;
    pthread_t thread;
    int victim_block_nr;

    // 버퍼캐시가 꽉참 -> victim 선정
    if (buffercache->items == CACHE_SIZE)
    {
        ret = -1;
        fprintf(stdout, "[ Write ]\tThere are no empty spaces.\n");
        char *data = (char *)malloc(BLOCK_SIZE);
        switch (mode)
        {
        case 0:
            victim_block_nr = fifo(buffercache, data);
            // printf("victim : %s\n", data);
            break;
        case 1:
            victim_block_nr = lru(buffercache, data);
            // printf("victim : %s\n", data);
            break;
            break;
        case 2:
            victim_block_nr = lfu(buffercache, data);
            // printf("victim : %s\n", data);
            break;
            break;
        default:
            return -1;
        }
        fprintf(stdout, "[ Write ]\tVictim block number is %d.\n", victim_block_nr);

        // 스레드 생성 -> direct_io
        Args *a = (Args *)malloc(sizeof(Args));
        a->victim_block_nr = victim_block_nr;
        strcpy(a->data, data);
        int tid = pthread_create(&thread, NULL, direct_io, (void *)a);
        if (tid < 0)
        {
            perror("thread creating failed");
            // return -1;
        }
        pthread_join(thread, NULL);
        free(a);
        free(data);
        // return -1;
    }
    else
        fprintf(stdout, "[ Write ]\tThere are empty spaces.\n");

    // buffer cache에 쓰기
    pthread_mutex_lock(&lock);

    fprintf(stdout, "[ Write ]\tBlock number: %d, data: \"%s\"\n", block_nr, input);
    struct block *blk = (struct block *)malloc(sizeof(struct block));
    blk->block_nr = block_nr;
    strcpy(blk->data, input);
    blk->dirty_bit = 1;
    blk->ref_count = 0;
    fprintf(stdout, "\t\tDirty bit of Block number %d is now turned on.\n", block_nr);
    Node *node = (Node *)malloc(sizeof(Node));
    node->blk = blk;
    node->next = NULL;

    int index = hash(block_nr);
    fprintf(stdout, "[ HASH ]\tBlock number %d has index %d\n", block_nr, index);

    Node *current = buffercache->array[index];
    if (current == NULL)
    {
        buffercache->array[index] = node;
    }
    else
    { // not NULL
        while (current->next != NULL)
        {
            current = current->next;
        }
        current->next = node;
    }
    buffercache->items++;
    enqueue(buffercache->cachequeue, block_nr); // block_nr를 큐에 집어넣음
    push(buffercache->cachstack, block_nr);     // block_nr을 스택에 집어넣음

    pthread_mutex_unlock(&lock);
    return ret;
}
```

1) 버퍼 캐시가 꽉 찼을 때

1. Replacement Policy 알고리즘으로 victim block 선정
2. Args 구조체를 생성하고 스레드를 생성하여 direct_io 함수를 실행

2) 버퍼 캐시에 쓰기

i. block 타입의 동적 메모리를 할당

ii. 할당한 블록에 주어진 블록 번호를 설정 -> 주어진 입력 데이터를 블록에 복사

iii. 블록의 dirty 비트를 1로 설정, 블록의 참조 횟수를 0으로 초기화

iv. Current가 NULL 인 경우: 현재 주어진 해시 인덱스에 해당하는 연결 리스트가 비어있음 -> 새로운 노드를 첫 번째 노드로 설정

v. Current가 NULL 이 아닌 경우: 연결 리스트의 끝까지 이동하여 마지막 노드를 찾음 -> 마지막 노드의 다음에 새로운 노드를 추가

vi. 버퍼 캐시의 항목 수를 증가

vii. 큐에 현재 블록 추가[FIFO], 스택에 현재 블록 push [LRU]

A. direct_io

```c
// 스레드가 실행할 함수
void *direct_io(void *ptr)
{
    pthread_mutex_lock(&lock); // lock
    // int disk_fd = open("diskfile", O_RDWR|O_DIRECT);
    Args *args = (Args *)ptr;
    int block_nr = args->victim_block_nr;
    char *data = args->data;
    fprintf(stdout, "[ DIRECT I/O ]\tData: \"%.20s...\"\n", data);
    if (lseek(disk_fd, block_nr * BLOCK_SIZE, SEEK_SET) < 0)
        perror("disk is not mounted");
    if (write(disk_fd, data, BLOCK_SIZE) < 0)
    {
        perror("write - direct_io");
        pthread_mutex_unlock(&lock);
        return NULL;
    }
    fprintf(stdout, "[ DIRECT I/O ]\tBlock number %d has been written on disk.\n", block_nr);
    // close(disk_fd);
    pthread_mutex_unlock(&lock); // unlock
    return NULL;
}
```

주어진 데이터를 디스크의 특정 블록에 직접 쓰는 작업을 수행하는 함수

B. flush()

```
void *flush()
{ // void *ptr) {
    // BufferCache *bc = (BufferCache *) ptr;
    fprintf(stdout, "\n[ FLUSH ]\tFlushing thread is running...\n");
    while (exit_flag == 0)
    {
        for (int i = 0; i < CACHE_SIZE; i++)
        {
            Node *current = bc->array[i];
            while (current != NULL)
            {
                if (current->blk->dirty_bit == 1)
                {
                    if (exit_flag != 0)
                        return NULL;
                    pthread_mutex_lock(&lock); // lock
                    current->blk->dirty_bit = 0;
                    int block_nr = current->blk->block_nr;
                    fprintf(stdout, "[ FLUSH ]\tDirty bit of Block number %d is now turned off.\n", block_nr);
                    char *data = current->blk->data;
                    fprintf(stdout, "[ FLUSH ]\tData: \"%.10s...\"\n", data);
                    // int disk_fd = open("diskfile", O_RDWR|O_DIRECT);
                    if (lseek(disk_fd, block_nr * BLOCK_SIZE, SEEK_SET) < 0)
                        perror("disk is not mounted");
                    if (write(disk_fd, data, BLOCK_SIZE) < 0)
                        perror("write - flush");
                    pthread_mutex_unlock(&lock); // unlock
                }
                current = current->next;
            }
        }
        sleep(15);
    }
    return NULL;
}
```

1) 버퍼 캐시의 모든 블록을 확인하면서 dirty_bit이 1인 블록을 찾는다

2) 해당 블록의 dirty_bit을 0으로 설정, 데이터를 디스크에 플러시

3) 15초 동안 대기한 후 루프를 반복

4) exit_flag가 0이 아니면 함수를 종료

3. Replacement Policy

A. FIFO 알고리즘

```c
int fifo(BufferCache *bc, char *placeholder)
{
    Queue *q = bc->cachequeue;
    int block_nr = dequeue(q);
    int index = hash(block_nr);

    fprintf(stdout, "[ FIFO ]\tBlock number %d will be deleted.\n", block_nr);

    Node *current = bc->array[index];

    if (current == NULL) {
        return -1;
    }
    // victim이 첫번째 노드인 경우
    if (current->blk->block_nr == block_nr && current->next == NULL)
    {
        strcpy(placeholder, current->blk->data);
        free(current->blk);
        bc->array[index] = NULL;
        bc->items--;
        // 찾은 블록의 번호를 스택에서 삭제
        remove_value_from_stack(bc->cachstack, block_nr);
        return block_nr;
    } else if (current->blk->block_nr == block_nr) {
        strcpy(placeholder, current->blk->data);
        free(current->blk);
        bc->array[index] = current->next;
        bc->items--;
        return block_nr;
    }
    else
    {
        while (current->next != NULL)
        {
            // found
            if (current->next->blk->block_nr == block_nr)
            {
                strcpy(placeholder, current->blk->data);
                free(current->next->blk);
                current->next = NULL;
                bc->items--;
                // 찾은 블록의 번호를 스택에서 삭제
                remove_value_from_stack(bc->cachstack, block_nr);

                return block_nr;
            }
            current = current->next;
        }
    }
    perror("[ ERROR ]\tEntry Not found");
    return -1;
}
```

1) 큐에서 가장 먼저 들어온 블록 번호를 가져온다 (dequeue() -> victim block number
2) 가져온 블록 번호를 이용하여 버퍼 캐시 내의 어떤 인덱스에 위치하는지 계산한다
3) 주어진 해시 인덱스에 해당하는 연결 리스트의 첫 번째 노드를 가져온다
4) 첫 번째 노드가 삭제 대상인 경우
    i. 첫 번째 노드의 블록 번호가 삭제 대상과 일치하면, 삭제할 블록의

데이터를 placeholder에 복사한다

ii. 버퍼 캐시의 해당 인덱스의 첫 번째 노드를 NULL로 설정하여 리스트에서 제거한다

iii. 버퍼 캐시의 항목 수를 감소시킨다

5) 삭제 대상이 첫 번째 노드가 아닌 경우

i. 연결 리스트의 끝까지 이동한다

ii. 다음 노드의 블록 번호가 삭제 대상과 일치하면 삭제할 블록의 데이터를 placeholder에 복사한다

iii. 현재 노드의 다음 노드를 NULL로 설정하여 리스트에서 제거한다

iv. 버퍼 캐시의 항목 수를 감소한다

6) 찾은 Victim block stack에서 삭제

B. LRU 알고리즘

```c
int lru(BufferCache *buffercache, char *placeholder)
{
    int oldest_block_nr = -1;

    if (buffercache->cachstack->top >= 0)
    {
        // 가장 오래된 블록은 스택의 젤 아래있음
        oldest_block_nr = buffercache->cachstack->items[0];
    }
    fprintf(stdout, "[ LRU ]\tBlock number %d will be deleted.\n", oldest_block_nr);

    if (oldest_block_nr != -1)
    {
        // 스택에서 victim 제거
        remove_value_from_stack(buffercache->cachstack, oldest_block_nr);

        int index = hash(oldest_block_nr);
        Node *current = buffercache->array[index];
         if (current == NULL) {

        return -1;
    }

        if (current->blk->block_nr == oldest_block_nr)
        {
            strcpy(placeholder, current->blk->data);
            free(current->blk);
            bc->array[index] = NULL;
            bc->items--;
            // dequeue(buffercache->cachequeue);
            remove_value_from_queue(buffercache->cachequeue, oldest_block_nr);
            return oldest_block_nr;
        }
        else
        {
            while (current->next != NULL)
            {
                // found
                if (current->next->blk->block_nr == oldest_block_nr)
                {
                    strcpy(placeholder, current->blk->data);
                    free(current->next->blk);
                    current->next = NULL;
                    bc->items--;
                    remove_value_from_queue(buffercache->cachequeue, oldest_block_nr);
                    return oldest_block_nr;
                }
                current = current->next;
            }
        }
    }

    perror("Entry Not found");
    return -1;
}
```

1) 스택이 비어 있지 않은 경우 스택의 가장 아래에 있는 블록을 가장 오래된 블록으로 설정

2) 스택에서 가장 오래된 블록 (cachstack->items[0]) 삭제

3) 오래된 블록 번호 즉 Victim block 번호 이용하여 버퍼 캐시 내의 어떤 인덱스에 위치하는지 계산

4) 주어진 해시 인덱스에 해당하는 연결 리스트의 첫 번째 노드를 가져온다

5) 첫 번째 노드가 삭제 대상인 경우

   i. 첫 번째 노드의 블록 번호가 삭제 대상과 일치하면, 삭제할 블록의 데이터를 placeholder에 복사

ii. 버퍼 캐시의 해당 인덱스의 첫 번째 노드를 NULL로 설정하여 리스트에서 제거

iii. 버퍼 캐시의 항목 수를 감소

6) 삭제 대상이 첫 번째 노드가 아닌 경우

i. 연결 리스트의 끝까지 이동한

ii. 다음 노드의 블록 번호가 삭제 대상과 일치하면 삭제할 블록의 데이터를 placeholder에 복사

iii. 현재 노드의 다음 노드를 NULL로 설정하여 리스트에서 제거

iv. 버퍼 캐시의 항목 수를 감소

7) Victim block queue에서 삭제

8) Victim block stack에서 삭제

C. LFU 알고리즘

```c
int lfu(BufferCache *buffercache, char *placeholder)
{
    int min_ref_count = 10000;
    int victim_block_nr = -1;

    for (int i = 0; i < CACHE_SIZE; i++)
    {
        Node *current = buffercache->array[i];
        while (current != NULL)
        {
            if (current->blk->ref_count < min_ref_count)
            {
                min_ref_count = current->blk->ref_count;
                victim_block_nr = current->blk->block_nr;
            }
            current = current->next;
        }
    }
    fprintf(stdout, "[ LFU ]\tBlock number %d will be deleted.\n", victim_block_nr);
    // Check if victim found
    if (victim_block_nr != -1)
    {
        int index = hash(victim_block_nr);

        Node *current = buffercache->array[index];
         if (current == NULL) {
        printf(" !!!!!! Segmentation Fault occurs because current is NULL !!!!!!!!!!\n");
        return -1;
        }

        if (current->blk->block_nr == victim_block_nr)
        {
            strcpy(placeholder, current->blk->data);
            free(current->blk);
            bc->array[index] = NULL;
            bc->items--;
           remove_value_from_queue(buffercache->cachequeue, victim_block_nr);
           remove_value_from_stack(buffercache->cachstack, victim_block_nr);
            return victim_block_nr;
        }
        else
        {
            while (current->next != NULL)
            {
                // found
                if (current->next->blk->block_nr == victim_block_nr)
                {
                    strcpy(placeholder, current->blk->data);
                    free(current->next->blk);
                    current->next = NULL;
                    bc->items--;
                   remove_value_from_queue(buffercache->cachequeue, victim_block_nr);
                    remove_value_from_stack(buffercache->cachstack, victim_block_nr);
                    return victim_block_nr;
                }
                current = current->next;
            }
        }
    }

    perror("Entry Not found");
    return -1;
}
```

1) Ref_count가 가장 적은 블록을 victim block 으로 선정
2) Victim block 번호를 이용하여 버퍼 캐시 내의 어떤 인덱스에 위치하는지 계산
3) 주어진 해시 인덱스에 해당하는 연결 리스트의 첫 번째 노드를 가져온다
4) 첫 번째 노드가 삭제 대상인 경우
    i. 첫 번째 노드의 블록 번호가 삭제 대상과 일치하면, 삭제할 블록의

데이터를 placeholder에 복사합니다.

ii. 버퍼 캐시의 해당 인덱스의 첫 번째 노드를 NULL로 설정하여 리스트에서 제거

iii. 버퍼 캐시의 항목 수를 감소

5) 삭제 대상이 첫 번째 노드가 아닌 경우

i. 연결 리스트의 끝까지 이동

ii. 다음 노드의 블록 번호가 삭제 대상과 일치하면 삭제할 블록의 데이터를 placeholder에 복사

iii. 현재 노드의 다음 노드를 NULL로 설정하여 리스트에서 제거

iv. 버퍼 캐시의 항목 수를 감소

6) Victim block queue에서 삭제

7) Victim block stack에서 삭제

3.1.2.2 Buffer cache 검증 결과

1. Test.c 구조

1) init() 함수 호출 -> 초기화

2) 명령행 인수를 통해 지정된 대체 알고리즘을 설정

3) buffer_init() 함수 호출 -> 버퍼 초기화

```
for (int i = 0; i < diskfile_block; i++) {
    char *temp;
    temp = malloc(BLOCK_SIZE);
    sprintf(temp, "Initial data: block number %d", i);
    os_write(i, temp);
    free(temp);
}
```

4) os_write 함수를 사용하여 diskfile_block 개수만큼 초기 데이터를 디스크에 기록

```
fprintf(stdout, "Initial access sequence: ["); fflush(stdout);
int seq_init[CACHE_SIZE] = {3, 19, 25, 14, 11, 5, 8, 54, 36, 28, 13, 56, 76, 61, 43, 52, 85, 40, 60};
srand(time(NULL)+t); t++;
for (int i = 0; i < buffer_block-1; i++) {
    fprintf(stdout, " %d", seq_init[i]); fflush(stdout);
}
fprintf(stdout, " ]\n");

for (int i = 0; i < buffer_block-1; i++) {
    char *data = (char *)malloc(strlen(sample) + 10);
    strcpy(data, sample);
    char number[5];
    sprintf(number, "%d", seq_init[i]);
    strcat(data, number);

    int temp = delayed_write(bc, seq_init[i], data, algorithm_index);
    free(data);
    if (temp != 0)
        fprintf(stdout, "Write faild! Block numder: %d\n", seq_init[i]);
    else {
        char *temp_buffer = malloc(BLOCK_SIZE);
        buffered_read(bc, seq_init[i], temp_buffer);
        free(temp_buffer);
    }
}
```

5) 미리 정의된 초기 access sequence seq_init을 사용, delayed_write 호출하여 버퍼 캐시에 데이터 기록

```
//flushing thread init
if (pthread_create(&flushing_thread, NULL, flush, (void *)bc) != 0) {
    perror("thread creating failed");
}
pthread_detach(flushing_thread);
```

6) pthread_create 함수 -> flushing 스레드를 생성 pthread_detach -> 생성된 스레드를 분리

7) 무작위로 생성된 액세스 시퀀스를 사용하여 os_read 함수를 호출하여 버

퍼 캐시에서 블록을 읽고 hit ratio 기록

8) 무작위로 생성된 액세스 시퀀스를 사용하여 delayed_write 함수를 호출하여 버퍼 캐시에 write 및 disk_read를 이용하여 디스크에서 블록을 읽기

2. 결과

```
Buffer Cache Simulator Test
---------------------------------------------------------------
Initialize

Initial access sequence: [ 3 19 25 14 11 5 8 54 36 28 13 56 76 61 43 52 85 40 6
0 ]
[ Write ]       There are empty spaces.
[ Write ]       Block number: 3, data: "sample#3"
                Dirty bit of Block number 3 is now turned on.
[ HASH ]        Block number 3 has index 3
[ Hit ]         Block numder: 3, data: "sample#3..." time: 2 ms
[ Write ]       There are empty spaces.
[ Write ]       Block number: 19, data: "sample#19"
                Dirty bit of Block number 19 is now turned on.
[ HASH ]        Block number 19 has index 19
[ Hit ]         Block numder: 19, data: "sample#19..." time: 1 ms
[ Write ]       There are empty spaces.
[ Write ]       Block number: 25, data: "sample#25"
                Dirty bit of Block number 25 is now turned on.
[ HASH ]        Block number 25 has index 5
[ Hit ]         Block numder: 25, data: "sample#25..." time: 2 ms
```

미리 정의된 초기 access sequence 로 buffer cache에 write

Buffer cache의 구조가 hash table 임을 보여줌

```
Buffered Read

Access sequence: [ 15 11 3 19 0 4 13 4 11 2 15 1 16 8 13 18 8 16 18 2 13 15 4
 5 12 2 17 9 7 18 ]
[ Miss ]        Block numder: 15, time: 1 ms
[ Hit ]         Block numder: 11, data: "sample#11..." time: 2 ms
```

무작위로 생성된 액세스 시퀀스를 사용하여 os_read 를 호출

A. FIFO

```
[ Miss ]        Block numder: 12, time: 1 ms
[ Miss ]        Block numder: 2, time: 1 ms
[ Miss ]        Block numder: 17, time: 1 ms
[ Miss ]        Block numder: 9, time: 1 ms
[ Miss ]        Block numder: 7, time: 1 ms
[ Miss ]        Block numder: 18, time: 1 ms
[Read] hit ratio : 36.67%
```

Buffer read 시 Hit, miss 소요 시간 기록 및 hit ratio 기록

FIFO read hit ratio **36.67%**

```
Delayed Write

Replacement Algorithm: FIFO
Access sequence: [ 11 13 19 8 4 6 1 11 5 9 7 8 12 8 8 7 3 6 13 12 14 0 1 2 5
17 18 15 0 17 ]
```

무작위로 생성된 액세스 시퀀스를 사용하여 delayed_write 를 호출

```
[ Write ]       Block number: 19, data: "#19 sample FIFO data"
                Dirty bit of Block number 19 is now turned on.
[ HASH ]        Block number 19 has index 19
[ DISK READ ]   Block number 19 data: "sample#19..."
[ Write ]       There are no empty spaces.
Queue : [ -1 25 14 11 5 8 54 36 28 13 56 76 61 43 52 85 40 60 11 13 19 ]
[ FIFO ]        Block number 25 will be deleted.
[ Write ]       Victim block number is 25.
```

delayed_write 및 buffer가 찼을때 FIFO를 위해 정의한 queue에서 victim select

```
[ FLUSH ]       Dirty bit of Block number 28 is now turned off.
[ FLUSH ]       Data: "sample#28..."
```

Back ground에서 flush 스레드 동작 확인 [FIFO, LRU, LFU 공통]

B. LRU

```
[ Miss ]        Block numder: 12, time: 1 ms
[ Hit ]         Block numder: 11, data: "sample#11..." time: 1 ms
[Read] hit ratio : 43.33%
```

LRU read hit ratio **43.33%**

```
Delayed Write

Replacement Algorithm: LRU
Access sequence: [ 2 5 7 1 2 1 12 0 16 17 7 18 9 11 3 14 18 13 11 1 12 4 13 6
 1 8 1 11 9 6 ]
[ Write ]       There are empty spaces.
[ Write ]       Block number: 2, data: "#2 sample LRU data"
                Dirty bit of Block number 2 is now turned on.
[ HASH ]        Block number 2 has index 2
[ DISK READ ]   Block number 2 data: "..."
[ Write ]       There are no empty spaces.
Stack contents: 25 5 54 36 28 56 76 61 43 52 85 40 60 8 19 3 14 13 11 2
[ LRU ] Block number 25 will be deleted.
[ Write ]       Victim block number is 25.
```

무작위로 생성된 액세스 시퀀스를 사용하여 delayed_write 를 호출 및 buffer가 찼을때 LRU를 위해 정의한 stack 에서 victim select (contents 뒤쪽이 top)

C. LFU

```
[ Miss ]        Block numder: 18, time: 1 ms
[ Miss ]        Block numder: 0, time: 1 ms
[ Hit ]         Block numder: 3, data: "sample#3..." time: 1 ms
[ Hit ]         Block numder: 5, data: "sample#5..." time: 2 ms
[ Hit ]         Block numder: 14, data: "sample#14..." time: 2 ms
[Read] hit ratio : 53.33%
```

LRU read hit ratio **53.33%**

```
[ Write ]       Block number: 6, data: "#6 sample LFU data"
                Dirty bit of Block number 6 is now turned on.
[ HASH ]        Block number 6 has index 6
[ DISK READ ]   Block number 6 data: "sample#14..."
[ Write ]       There are no empty spaces.
[ LFU ] ref count is0
[ LFU ] Block number 6 will be deleted.
[ Write ]       Victim block number is 6.
[ DIRECT I/O ]  Data: "#6 sample LFU data..."
```

무작위로 생성된 액세스 시퀀스를 사용하여 delayed_write 를 호출 및 buffer가 찼을때 LFU를 위해 정의한 ref_count 를 이용하여 victim select

(block 6은 바로 직전에 write 되었기 때문에 ref_count 가 0임 -> victim)