학번 : 2020540023
학과 : 수학과
이름 : 이명규

1. 소스코드 전체와 설명

- 소스코드의 흐름

작성한 소스코드의 흐름은 대략 아래와 같고, 자세한 설명은 코드에 주석을 하였습니다.

```
<FIFO>
while (not EOF)                    // 파일 끝까지
    if (blkno not in hash_table)   // blkno가 hash_table 안에 없으면 (캐시 미스)
        if is_full()               // 캐시 블록을 모두 사용중이면
            delete()               // 가장 예전에 들어온 캐시 블록 삭제
        insert()                   // 입력된 데이터 번호를 캐시 블록에 기록
        miss <- miss + 1           // miss 1 증가
    else                           // blkno가 hash_table 안에 있으면 (캐시 히트)
        hit <- hit + 1             // hit 1 증가

<LRU>
while (not EOF)                    // 파일 끝까지
    if (blkno in hash_table)       // blkno가 hash_table 안에 있으면 (캐시 히트)
        update()                   // blkno 캐시 블록을 가장 최근에 참조 -> 업데이트
        hit <- hit + 1             // 캐시 히트 (hit 1 증가)
    else                           // blkno가 hash_table 안에 없으면 (캐시 미스)
        if is_full()               // 캐시 블록을 모두 사용중이면
            delete()               // 가장 예전에 사용된 캐시 블록 삭제
        insert()                   // 입력된 데이터 번호를 캐시 블록에 기록
        miss <- miss + 1           // miss 1 증가
```

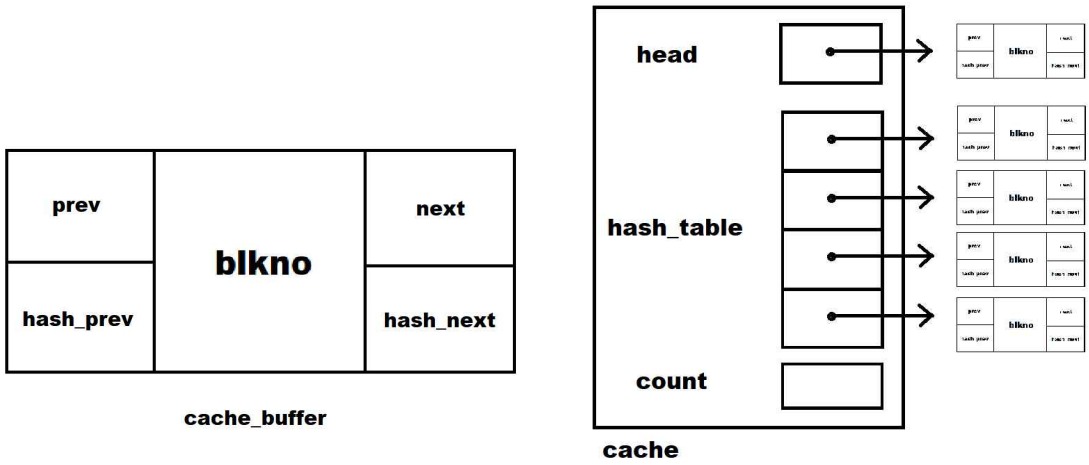
- 사용된 함수의 설명

위에서 사용된 함수의 설명은 각각 아래와 같습니다.

1. is_full() 함수는 캐시 블록 8192가 모두 사용중인지 확인하는 함수이고 TRUE(1), FALSE(0)을 반환합니다.
2. delete() 함수는 FIFO와 LRU에서의 작동이 다릅니다. FIFO 알고리즘에서는 현재 존재하는 캐시 블록 중 가장 예전에 입력된 캐시 블록을 삭제하는 반면에, LRU 알고리즘에서는 가장 예전에 사용된 캐시 블록을 삭제합니다.
3. insert() 함수는 FIFO 리스트와 LRU 리스트의 가장 마지막(최근에 사용됨)에 삽입하는 연산을 수행합니다.
4. update() 함수는 LRU 알고리즘의 구현에서만 사용되었으며, 가장 최근에 사용된 캐시 블록을 해당 위치에서 삭제하고 가장 마지막(최근에 사용됨)에 삽입하는 연산을 수행합니다.

- 구조체 설명

사용한 구조체 각각의 구조는 다음과 같습니다.



1. cache_buffer 구조체의 경우, '데이터 번호를 기록하는 blkno 필드(데이터 필드 역할)', 'FIFO와 LRU 리스트 관리를 위한 prev와 next 필드(링크 필드 역할)', '해시 테이블의 각 버켓에서 체이닝 기법을 적용할 수 있게 해주는 hash_prev와 hash_next 필드(링크 필드 역할)'로 구성되어 있습니다.

2. cache 구조체의 경우, 'FIFO와 LRU 리스트의 이중연결리스트를 관리하게 해주는 헤드노드(캐시 블록 구조체)를 가리키는 head 필드', '캐시 블록을 가리키는 포인터 배열로 이루어진 hash_table 필드', '사용되고 있는 캐시 블록의 수를 기록하는 count 필드'로 구성되어 있습니다.

- 구조체의 초기화 과정

각 구조체는 다음과 같이 초기화를 해주었습니다.

1. cache_buffer 구조체에서 blkno는 -1로(-1이면 헤드노드), 네 가지 링크 필드는 모두 자신을 가리키도록 초기화하였습니다.

2. cache 구조체에서 head는 이중연결리스트를 관리할 헤드노드를 생성하여 초기화한 후, 해당 노드를 가리키도록 하였습니다. hash_table 역시 각각의 버켓을 담당하는 이중연결리스트의 헤드노드를 생성하여 초기화한 후, 배열의 각 요소에 해당하는 포인터들이 이 헤드노드들을 가리키도록 하였습니다. 그리고 count는 0으로 초기화하였습니다. 앞서 설명한 cache 구조체의 초기화 과정을 위의 그림 중 오른쪽 그림을 통해 표현하였습니다.

- 소스코드 공유 (링크)

마지막으로, 문서에 복사한 소스코드 이외에도, 가독성을 위해서 코드를 공유하는 사이트를 통해서 코드를 공유합니다. 아래의 링크에 들어가시면, 문서에 작성한 것과 동일한 코드를 좀 더 읽기 편한 환경에서 확인하실 수 있습니다.

FIFO : https://colorscripter.com/s/IW2Myad

LRU : https://colorscripter.com/s/On5kB4K

- 전체 소스코드

(1) FIFO

```
#include <stdio.h>
#include <stdlib.h>

#define CACHE_SIZE 8192   // 캐시 크기(캐시 블록의 개수)
#define TABLE_SIZE 4001   // 해싱 테이블의 크기
#define TRUE 1
#define FALSE 0

typedef unsigned longelement;
typedef structcache_buffer {
    element blkno;          // 데이터 번호 in 참조 스트림
    structcache_buffer *next, *prev;
    structcache_buffer *hash_next, *hash_prev;
} Cache_Buffer;
typedef structcache {      // 구조체 선언
    Cache_Buffer *head;     // FIFO를 위한 헤드노드 (큐처럼, 삽입된 순서대로 노드 관리)
    Cache_Buffer *hash_table[TABLE_SIZE];   // 해시 테이블 선언
    intcount;              // 전체 캐시블록의 수(8192개가 넘었는지 체크하기 위함)
} Cache;

// 캐시 블록을 초기화: 헤드노드의 링크들이 자신을 가리키도록 함
voidinit_block(Cache_Buffer *p)
{
    p->blkno = -1;      // 헤드 노드일 경우 -1로 초기화하기 위함
    p->next = p;
    p->prev = p;
    p->hash_next = p;
    p->hash_prev = p;
}

// Cache 구조체를 초기화
voidinit(Cache *c)
{
    // 헤드 초기화
    c->head = (Cache_Buffer *)malloc(sizeof(Cache_Buffer));
    init_block(c->head);
    // 해시 테이블 초기화
    for(inti = 0; i < TABLE_SIZE; i++) {
        (c->hash_table)[i] = (Cache_Buffer *)malloc(sizeof(Cache_Buffer));
        init_block((c->hash_table)[i]);
    }
    // count 초기화
    c->count = 0;
```

```
}

// 캐시 블록이 8192개 모두 쓰였는지 확인
intis_full(Cache *c)
{
    return(c->count == CACHE_SIZE);
}

// 제산 함수를 사용한 해싱 함수
inthash_function(intkey)
{
    returnkey % TABLE_SIZE;
}

// 체인법을 이용하여 테이블에서 키의 존재 여부 검사
intis_hash_in(Cache_Buffer *ht[], element num)
{
    inthash_value = hash_function(num);
    Cache_Buffer *bucket = ht[hash_value];
    Cache_Buffer *node;
    if(bucket->hash_next == bucket)    // 헤드노드만 존재하는 경우(해당 버킷의 슬롯이 0개)
        return FALSE;
    for(node = bucket->hash_next; node != bucket; node = node->hash_next)
        if(node->blkno == num)
            return TRUE;
    return FALSE;
}

// (존재 X 보장 후)cache 구조체에서 새로운 데이터를 해시 테이블에 삽입
voidinsert(Cache *c, element num)
{
    inthash_value = hash_function(num);    // 해시함수에 값 대입
    Cache_Buffer **ht = c->hash_table;
    Cache_Buffer *phead = c->head;
    // 삽입할 캐시 블록(노드) 생성
    Cache_Buffer *p;
    p = (Cache_Buffer *)malloc(sizeof(Cache_Buffer));
    p->blkno = num;
    // 해시 관련 링크 관리
    p->hash_prev = ht[hash_value]->hash_prev;
    p->hash_next = ht[hash_value];
    ht[hash_value]->hash_prev->hash_next = p;
    ht[hash_value]->hash_prev = p;
    // FIFO 위한 링크 관리
    p->prev = phead->prev;
    p->next = phead;
```

```
    phead->prev->next = p;
    phead->prev = p;
    // count 증가
    c->count++;
}

// 가장 예전에 삽입된 노드 삭제 for FIFO 리스트 관리
voiddelete(Cache *c)
{
    Cache_Buffer *removed = c->head->next;
    // FIFO 위한 링크 관리
    removed->prev->next = removed->next;
    removed->next->prev = removed->prev;
    // 해시 링크 관리
    removed->hash_prev->hash_next = removed->hash_next;
    removed->hash_next->hash_prev = removed->hash_prev;
    // count 감소
    c->count--;
    // romoved 할당 해제
    free(removed);
}

intmain()
{
    inthit = 0, miss = 0;  // 캐시 히트율 측정용
    unsigned longnum;
    Cache c;
    init(&c);

    FILE *fp;
    fp = fopen("test_trace.txt", "r");
    if(fp == NULL)
    {
        fprintf(stderr, "열 수 없음.\n");
        exit(1);
    }

    // 정상적으로 파일 열고난 후
    while(!feof(fp))
    {
        fscanf(fp, "%d", &num);             // 파일에서 입력 받기
        if(!is_hash_in(c.hash_table, num)) // 해시테이블에 존재하지 않으면..(존재하면 조회
하고 끝)
        {
            if(is_full(&c))               // 가득 찼으면
                delete(&c);               // 가장 예전에 삽입된 노드 삭제
```

```
            insert(&c, num);                // 입력된 데이터 삽입(하고 조회..)
            miss++;
        }
        else
            hit++;
    }
    fclose(fp);

    inttotal = hit + miss;
    floathit_ratio = (float)hit/(hit+miss), miss_ratio = (float)miss/(hit+miss);
    printf("hit ratio = %f, miss ratio = %f\n", hit_ratio, miss_ratio);
    printf("\n");
    printf("tatal access = %d, hit = %d, miss = %d\n", total, hit, miss);
    printf("Hit ratio = %f\n", hit_ratio);
    return 0;
}
```

(2) LRU

```
#include <stdio.h>
#include <stdlib.h>

#define CACHE_SIZE 8192   // 캐시 크기(캐시 블록의 개수)
#define TABLE_SIZE 4001   // 해싱 테이블의 크기
#define TRUE 1
#define FALSE 0

typedef unsigned longelement;
typedef structcache_buffer {
    element blkno;          // 데이터 번호 in 참조 스트림
    structcache_buffer *next, *prev;
    structcache_buffer *hash_next, *hash_prev;
} Cache_Buffer;
typedef structcache {      // 구조체 선언
    Cache_Buffer *head;     // FIFO를 위한 헤드노드 (큐처럼, 사용된 순서대로 노드 관리)
    Cache_Buffer *hash_table[TABLE_SIZE];   // 해시 테이블 선언
    intcount;              // 전체 캐시블록의 수(8192개가 넘었는지 체크하기 위함)
} Cache;

// 캐시 블록을 초기화: 헤드노드의 링크들이 자신을 가리키도록 함
voidinit_block(Cache_Buffer *p)
{
    p->blkno = -1;      // 헤드 노드일 경우 -1로 초기화하기 위함
    p->next = p;
    p->prev = p;
    p->hash_next = p;
    p->hash_prev = p;
}

// Cache 구조체를 초기화
voidinit(Cache *c)
{
    // 헤드 초기화
    c->head = (Cache_Buffer *)malloc(sizeof(Cache_Buffer));
    init_block(c->head);
    // 해시 테이블 초기화
    for(inti = 0; i < TABLE_SIZE; i++) {
        (c->hash_table)[i] = (Cache_Buffer *)malloc(sizeof(Cache_Buffer));
        init_block((c->hash_table)[i]);
    }
    // count 초기화
    c->count = 0;
}
```

```
// 캐시 블록이 8192개 모두 쓰였는지 확인
intis_full(Cache *c)
{
    return(c->count == CACHE_SIZE);
}

// 제산 함수를 사용한 해싱 함수
inthash_function(intkey)
{
    returnkey % TABLE_SIZE;
}

// 특정 값을 가지고 있는 노드(캐시 블록)를 검색해서 그 노드의 포인터를 반환하는 함수(존재 보장) -> update 함수에서 사용
Cache_Buffer* search(Cache_Buffer *ht[], element num)
{
    inthash_value = hash_function(num);
    Cache_Buffer *bucket = ht[hash_value];
    Cache_Buffer *node;
    if(bucket->hash_next == bucket)    // 헤드노드만 존재하는 경우(해당 버킷의 슬롯이 0개)
        return NULL;
    for(node = bucket->hash_next; node != bucket; node = node->hash_next)
        if(node->blkno == num)
            returnnode;
    return NULL;
}

// (존재 X 보장 후)cache 구조체에서 새로운 데이터를 해시 테이블에 삽입
voidinsert(Cache *c, element num)
{
    inthash_value = hash_function(num);    // 해시함수에 값 대입
    Cache_Buffer *phead = c->head;
    Cache_Buffer **ht = c->hash_table;
    // 삽입할 캐시 블록(노드) 생성
    Cache_Buffer *p;
    p = (Cache_Buffer *)malloc(sizeof(Cache_Buffer));
    p->blkno = num;
    // 해시 관련 링크 관리
    p->hash_prev = ht[hash_value]->hash_prev;
    p->hash_next = ht[hash_value];
    ht[hash_value]->hash_prev->hash_next = p;
    ht[hash_value]->hash_prev = p;
    // LRU 위한 링크 관리
    p->prev = phead->prev;
    p->next = phead;
```

```
    phead->prev->next = p;
    phead->prev = p;
    // count 증가
    c->count++;
}

// 가장 예전에 사용된 노드 삭제 for LRU 리스트 관리
voiddelete(Cache *c)
{
    Cache_Buffer *removed = c->head->next;
    // LRU 위한 링크 관리
    removed->prev->next = removed->next;
    removed->next->prev = removed->prev;
    // 해시 링크 관리
    removed->hash_prev->hash_next = removed->hash_next;
    removed->hash_next->hash_prev = removed->hash_prev;
    // count 감소
    c->count--;
    // romoved 할당 해제
    free(removed);
}

// (현재 존재하는 블록에 접근 시)사용했음을 업데이트함
voidupdate(Cache *c, element num)
{
    inthash_value = hash_function(num);
    Cache_Buffer *phead = c->head;
    Cache_Buffer **ht = c->hash_table;
    Cache_Buffer *access = search(ht, num);
    // 이전 링크들 삭제
    access->prev->next = access->next;
    access->next->prev = access->prev;
    // 링크 새롭게 연결
    access->prev = phead->prev;
    access->next = phead;
    phead->prev->next = access;
    phead->prev = access;
}

intmain()
{
    inthit = 0, miss = 0;  // 캐시 히트율 측정 위함
    unsigned longnum;
    Cache c;
    init(&c);
```

```
    FILE *fp;
    fp = fopen("test_trace.txt", "r");
    if(fp == NULL)
    {
        fprintf(stderr, "열 수 없음.\n");
        exit(1);
    }

    // 정상적으로 파일 열고난 후
    while(!feof(fp))
    {
        fscanf(fp, "%d", &num);         // 파일에서 입력 받기
        if(search(c.hash_table, num))  // 해시테이블에 존재한다면 최근에 사용했음을 업데이
트 for LRU
        {
            update(&c, num);
            hit++;
        }
        else                          // 존재하지 않으면..
        {
            if(is_full(&c))           // 가득 찼으면
                delete(&c);           // 가장 예전에 사용된 노드 삭제
            insert(&c, num);          // 입력된 데이터 삽입(하고 조회..)
            miss++;
        }
    }
    fclose(fp);

    inttotal = hit + miss;
    floathit_ratio = (float)hit/(hit+miss), miss_ratio = (float)miss/(hit+miss);
    printf("hit ratio = %f, miss ratio = %f\n", hit_ratio, miss_ratio);
    printf("\n");
    printf("tatal access = %d, hit = %d, miss = %d\n", total, hit, miss);
    printf("Hit ratio = %f\n", hit_ratio);
    return 0;
}
```

2. 시뮬레이션 결과 화면 캡쳐

3. LRU/FIFO 구현 및 시뮬레이션 결과보고

	FIFO	LRU
Hit ratio	76.4469%	77.8129%

실제 시뮬레이션을 돌린 결과, 캐시 히트율은 FIFO와 LRU 각각 76.4469%와 77.8129%로 교수님께서 알려주신 수치와 동일하게 나왔습니다. 또한, 실제로 LRU 교체 알고리즘이 FIFO의 히트율보다 높은 결과를 보인다는 것을 확인할 수 있었고, 따라서 데이터 접근 속도가 높으리라는 것을 예상할 수 있었습니다.