2017920017 컴퓨터과학부 김지윤

1. 소스코드 전체

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

#define MAX_SIZE 8192
#define SIZE 8192
#define FALSE -1
#define LRU 0
#define FIFO 1

typedef struct cache_buffer
{
    unsigned long blkno;
    struct cache_buffer *next, *prev;
    struct cache_buffer *hash_next, *hash_prev;
} cache_buffer;

int cache_hit = 0;
int count = 0;
int list_num = 0;

void hash_chain_print(cache_buffer *ht[]);
void print_list(cache_buffer *head);

void init_cache_list(cache_buffer *head)
{
    head->next = head;
    head->prev = head;
}

cache_buffer *delete_list(cache_buffer *head)
{
    cache_buffer *removed = head->prev;
    removed->prev->next = removed->next;
    removed->next->prev = removed->prev;

    return removed;
}

void insert_list_first(cache_buffer *head, cache_buffer *node)
{
    node->next = head->next;
    node->prev = head;
    head->next->prev = node;
    head->next = node;
```

```
}

void print_list(cache_buffer *head)
{
    for (cache_buffer *temp = head->next; temp != head; temp = temp->next)
        printf("%ld->", temp->blkno);
    printf("\n");
}

int hash_function(int key)
{
    return (key % MAX_SIZE);
}

int hash_chain_search(cache_buffer *ht[], cache_buffer *list_head, int blkno,
int type)
{
    int hash_value = hash_function(blkno);
    cache_buffer *head = ht[hash_value];
    cache_buffer *temp;

    for (temp = head->hash_next; temp != head; temp = temp->hash_next)
    {
        if (temp->blkno == blkno)
        {
            cache_hit++;
            if (type == LRU)
            {
                temp->prev->next = temp->next;
                temp->next->prev = temp->prev;
                insert_list_first(list_head, temp);
            }
            return blkno;
        }
    }
    return FALSE;
}

cache_buffer *hash_chain_add(cache_buffer *ht[], int blkno)
{
    int hash_value = hash_function(blkno);
    cache_buffer *head = ht[hash_value];
    cache_buffer *new = (cache_buffer *)malloc(sizeof(cache_buffer));
    new->hash_next = head->hash_next;
    new->hash_prev = head;
    new->blkno = blkno;
    head->hash_next->hash_prev = new;
    head->hash_next = new;
```

```
    return new;
}

void hash_chain_delete(cache_buffer *ht[], cache_buffer *removed)
{
    int blkno = removed->blkno;
    int hash_value = hash_function(blkno);

    removed->hash_prev->hash_next = removed->hash_next;
    removed->hash_next->hash_prev = removed->hash_prev;
    free(removed);
}

int hash_chain(cache_buffer *ht[], cache_buffer *list_head, int blkno, int
type)
{
    cache_buffer *new, *removed;
    int key = hash_chain_search(ht, list_head, blkno, type);

    if (key == FALSE)
    {
        new = hash_chain_add(ht, blkno);
        insert_list_first(list_head, new);

        if (list_num < SIZE)
        {
            list_num++;
        }
        else
        {
            removed = delete_list(list_head);
            hash_chain_delete(ht, removed);
        }
    }
    return key;
}

void init(cache_buffer *ht[])
{
    for (int i = 0; i < MAX_SIZE; i++)
    {
        cache_buffer *temp = malloc(sizeof(cache_buffer));
        temp->hash_next = temp;
        temp->hash_prev = temp;
        ht[i] = temp;
    }
}
```

```
void hash_chain_print(cache_buffer *ht[])
{
    cache_buffer *node;
    for (int i = 0; i < MAX_SIZE; i++)
    {
        printf("[%d]->", i);
        for (node = ht[i]->hash_next; node != ht[i]; node = node->hash_next)
            printf("%ld->", node->blkno);
        printf("\n");
    }
}

int main()
{
    cache_buffer *head = (cache_buffer *)malloc(sizeof(cache_buffer));
    cache_buffer *hash_table[MAX_SIZE];

    char buf[11];
    int buf_num;
    FILE *fp;
    float miss_ratio;
    float hit_ratio;

    init(hash_table);
    init_cache_list(head);

    if ((fp = fopen("./test_trace.txt", "r")) == NULL)
    {
        perror("Error");
        exit(-1);
    }

    /* test
        for (int i = 0; i < 500; i++)
        {
            count++;
            int k = i % 102;
            printf("[%d] ", k);
            hash_chain(hash_table, head, k, LRU);
        }

        hash_chain_print(hash_table);
        print_list(head);
        printf("\nresult : %d %d\n", cache_hit, count);
    */

    while ((fgets(buf, 11, fp)) > 0)
```

```
    {
        count++;
        buf_num = strtoul(buf, NULL, 10);
        hash_chain(hash_table, head, buf_num, FIFO);
    }

    printf("------------------------Result------------------------\n");

    printf("<FIFO>\n");

    hit_ratio = (float)cache_hit / count;
    miss_ratio = (float)(count - cache_hit) / count;
    printf("hit ratio : %f, miss ratio = %f\n\n", hit_ratio, miss_ratio);
    printf("total access = %d, ", count);
    printf("hit = %d, miss = %d\n", cache_hit, count - cache_hit);
    printf("Hit ratio = %f\n\n", hit_ratio);

    fseek(fp, 0L, SEEK_SET);
    count = 0;
    cache_hit = 0;
    list_num = 0;
    init_cache_list(head);
    init(hash_table);

    while ((fgets(buf, 11, fp)) > 0)
    {
        count++;
        buf_num = strtoul(buf, NULL, 10);
        hash_chain(hash_table, head, buf_num, LRU);
    }

    hit_ratio = (float)cache_hit / count;
    miss_ratio = (float)(count - cache_hit) / count;

    printf("<LRU>\n");
    printf("hit ratio : %f, miss ratio = %f\n\n", hit_ratio, miss_ratio);
    printf("total access = %d, ", count);
    printf("hit = %d, miss = %d\n", cache_hit, count - cache_hit);
    printf("Hit ratio = %f", hit_ratio);

    return 0;
}
```

<코드 설명>

hash table의 사이즈는 MAX_SIZE로, 버퍼 캐시의 사이즈는 SIZE로 지정합니다. 문제에서 정의

한 버퍼 캐시의 사이즈는 8192이므로 SIZE에는 8192, MAX_SIZE에는 임의의 수를 둘 수 있지만, 구현의 속도를 올리기 위해서 8192로 지정합니다. 구조체는 cache_buffer로 설정하였고, cache_buffer내에는 blkno와 캐시 리스트 내에서 사용하는 link인 next와 prev, 해시 테이블 내에서 사용하는 hash_next, hash_prev가 있습니다. 캐시 리스트는 현재 버퍼 캐시에 들어와 있는 캐시들의 리스트를 의미합니다. cache hit가 발생한 횟수를 cache_hit이라는 변수에 담아 저장하였고, 전체 탐색 횟수를 count, 캐시 리스트의 전체 갯수를 list_num 으로 저장합니다.

```
void init_cache_list(cache_buffer *head)
```
: 캐시 리스트를 초기화 하는 함수입니다.(헤드 노드)

```
cache_buffer *delete_list(cache_buffer *head)
```
: 교체(replacement)될 때, 캐시 리스트에서 제거하는 함수입니다, 리스트에서 순서는 뒤에 있을수록, 교체 우선순위가 높도록 순서를 유지할 것이므로 리스트의 맨 뒤에 있는 노드를 제거합니다.

```
void insert_list_first(cache_buffer *head, cache_buffer *node)
```
: LRU 방식이든, FIFO 방식이든 둘 다 삽입 시에 캐시 리스트의 맨 앞에 넣습니다. LRU, FIFO 방식 둘 다 뒤에 있을수록 교체 우선순위는 높습니다.

```
void hash_chain_print(cache_buffer *ht[]);
void print_list(cache_buffer *head);
```
: 테스팅을 위한 print 용도입니다.

```
int hash_function(int key)
```
: key를 전달하면, hash_value를 리턴하는 함수입니다.

```
int hash_chain_search(cache_buffer *ht[], cache_buffer *list_head, int blkno,
int type)
```
: 해시 테이블에서 blkno를 찾는 함수입니다. FIFO라면, 찾은 경우에 cache_hit만 증가시키고 blkno를 리턴하지만, LRU라면, 최근에 방문된 데이터는 캐시 리스트의 맨 앞으로 이동합니다. 연산 횟수가 상당히 많으므로, LRU를 0으로 정의해서 0이랑 비교하도록 연산을 구현해서, 속도를 조금 더 향상시키려고 하였습니다. 만약 hash table의 사이즈가 작다면, 해시 테이블의 각 인덱스마다 많은 수의 노드가 매달려 있으므로, 해시 테이블에서도 속도 감소를 위해서 재조정이 필요합니다. 하지만, 해시 테이블의 크기가 캐시 블록의 수와 비슷하게 설정했으므로 생략합니다. 조금

더 연산을 빠르게 하기 위해서 캐시 리스트의 순서 재조정 시 free후 다시 insert하지 않고, 순서만 바꿉니다.

```
cache_buffer *hash_chain_add(cache_buffer *ht[], int blkno)
```
: 해시 테이블에서 노드를 추가하는 함수입니다. Temporal locality를 고려해서, 최근에 들어온 건 처음에 넣을수록 좋을 테니, 리스트의 마지막에 넣는 것이 아닌 리스트의 처음에 넣습니다.

```
void hash_chain_delete(cache_buffer *ht[], cache_buffer *removed)
```
: 교체 정책(replacement policy)에 의해서 교체될 때, 해시 테이블에서 제거하는 함수입니다. 링크를 조정하고, 메모리에서 해제(free)합니다.

```
int hash_chain(cache_buffer *ht[], cache_buffer *list_head, int blkno, int type)
```
: 다른 교체 정책으로도 쉽게 구현할 수 있도록, 기능마다 하나의 함수로 구현하였고, hash_chain 함수에서 이를 통합합니다. 만약, hash function의 복잡도가 크다면, hash_value를 여기서 구해서 각 함수에 인수로 넘겨줄 수도 있겠지만, 모듈로 함수를 사용했으므로 blkno만 인수로 넘겨주었습니다. hash_chain_search 함수를 사용해서 blkno를 탐색한 뒤에, 찾았다면 리턴하고, 찾지 못했다면, 현재 버퍼 캐시에 없는 상황이므로 버퍼 캐시에 추가해야 합니다. 해시 테이블에 추가하고, 캐시 리스트에 추가합니다. List_num과 SIZE를 비교해, 아직 여유 공간이 있는 경우에는 list_num만 증가시키고, 더 이상 넣을 공간이 없다면 교체 정책에 의해서 리스트의 가장 마지막에 위치한 노드를 삭제합니다. (메모리 해제(free) 하지 않고 순서만 바꿉니다. 그 이유는, 해시 테이블에서도 순서를 조정해야 할 뿐 아니라, 이후의 hash_chain_delete 함수에서 인자로 넘겨받아서 다시 찾는 일이 없도록 하기 위해서 입니다.) 넘겨받은 가장 마지막에 위치한 노드를 해시테이블에서 삭제하고 메모리에서 해제(free) 합니다.

```
void init(cache_buffer *ht[])
```
: 해시 테이블을 초기화 해줍니다. (헤드 노드)

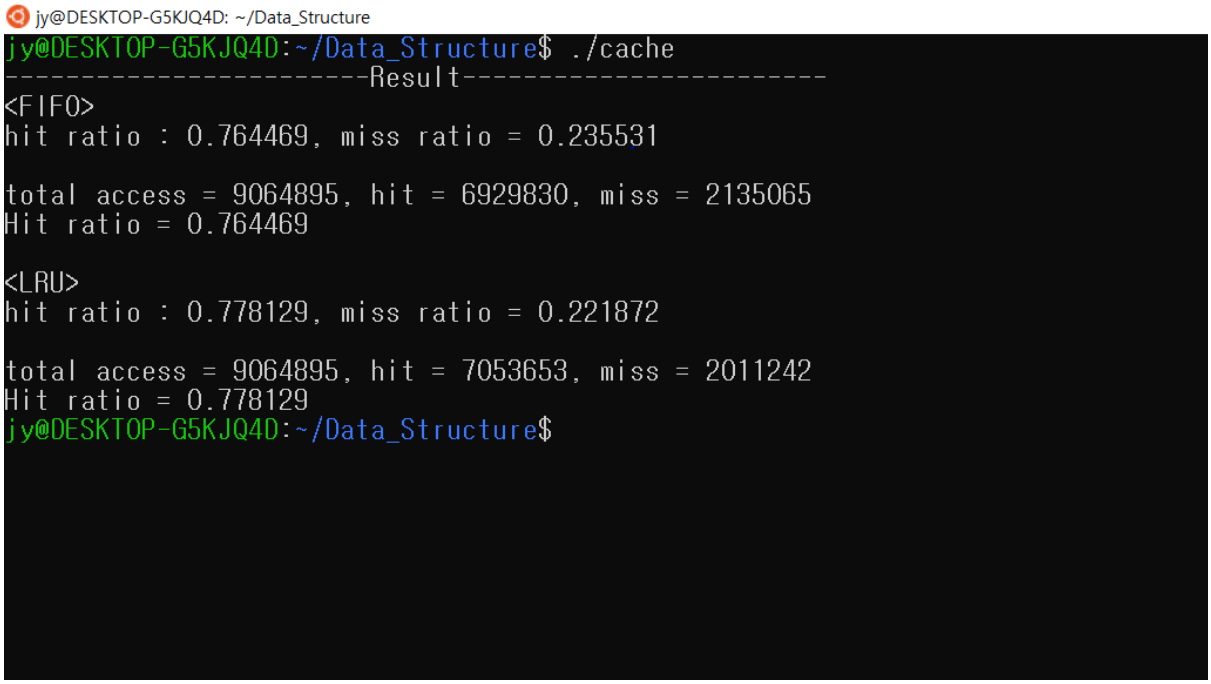
```
int main()
```
: main함수에서는, 해시 테이블과 캐시 리스트를 초기하고 각종 변수 들은 선언합니다. blkno는 unsigned long type이고, unsigned long type은 0~4294967295, 최대 10자리 수이므로, fgets를 통해서 넣는 buf의 크기는 11로 설정합니다. 그 이유는, fgets 함수는 Newline character 까지 읽고, 뒤에 NULL character를 추가시키므로, 최대 11이기 때문입니다. 읽는 양이 상당히 많으므로, 시스

템 콜 횟수를 줄이기 위해서, open이 아닌, fopen을 사용하였고, fopen 실패 시 프로그램은 종료합니다.

fgets함수의 리턴값이 0 초과라면, 1개의 수를 읽을 때마다 access를 한 것이므로, count 변수를 증가시키고, strtoul 함수를 이용해 buf를 unsigned long type으로 변환하고, hash_chain에 type을 FIFO로 지정한 뒤 함수를 반복해서 실행합니다.

이후에는 파일을 다시 열지 않고, fseek함수를 통해서 offset만 조정해줍니다. 그다음 count, cache_hit, list_num을 0으로 다시 설정하고, 해시테이블과 캐시리스트를 다시 초기화한 뒤 LRU방식으로 똑같이 호출합니다.

2. 시뮬레이션 결과 화면 캡처



3. LRU/FIFO 구현 및 시뮬레이션 결과 보고

	FIFO	LRU
Hit ratio	76.45%	77.81%

프로그램 수행 결과, LRU가 FIFO보다 더 좋은 성능을 보였습니다. hit ratio를 높이기 위해서는 지금 버퍼 캐시에 있는 블록이 다시 사용될 것인가가 제일 중요한 점입니다.

LRU가 FIFO보다 성능이 좋게 나타나는 이유는, 이 물음에 대해서, 최근에 들어왔다는 것보다는, 최근에 참조됐다는 것이 더 중요한 정보라는 것입니다. 물론, 여기에서는 blkno를 임의로 써서 만든 시뮬레이터이기 때문에 확정할 수는 없지만 LRU가 대부분의 상황에서 FIFO보다 수행능력이 좋습니다. 그 이유는 temporal locality와, 프로그램마다 자주 사용하는 Working Set을 계속 사용하기 때문입니다.

Locality에는 spatial locality와 temporal locality가 있는데, spatial locality는 현재 참조되는 자원과 인접한 곳에서 참조가 발생될 확률이 높다는 것입니다. 예를 들면 데이터베이스에서나, C 언어에서의 리스트, malloc을 통한 메모리 할당(heap에서의 연속적인 공간을 할당) 등이 있습니다. 인접한 곳에 다음 데이터가 있어서, 참조될 확률이 높습니다. temporal locality는 특정 시점에서 자원에 접근하면 그 자원은 다시 참조될 확률이 높은 것입니다. 예를 들면 반복문에서 x=list[0]라는 명령문이 수행되면, list의 data를 가지고 있는 block은 계속해서 참조될 것입니다. 또, 프로그램마다 자주 참조하고 사용하는 Working Set이 있고, 그 Working Set 내에 있는 데이터를 계속해서 참조할 것입니다. class, 객체에서 같은 필드, 같은 메소드를 계속 사용하는 것과 마찬가지 원리입니다.

만약 FIFO 방식의 경우라면, 데이터 베이스를 사용하는 경우, 마스터 인덱스를 포함한 인덱스 블록이 계속 교체될 것이고, 부팅 후 계속 사용해야 하는 중요한 블록들이 먼저 들어왔다는 이유만으로 버퍼 캐시에서 교체되면서 성능이 저하될 것입니다. LRU는 접근된 순서에 따라 정렬된 순서를 유지해야 하므로, FIFO 보다는 연산이 추가될 수는 있지만, 그보다 디스크 IO에 소모되는 비용이 훨씬 크기 때문에 cache hit ratio를 높이는 게 더 중요합니다.

물론 데이터를 어떤 순서로 어떻게 접근하는가, 동일 데이터를 얼마나 다시 읽는가, 다시 쓰는가에 따라 다르고, 모든 상황에 적합한 최선의 교체 정책은 없습니다. 예를 들면 데이터베이스에서는 한 번 읽고 다시 읽지 않는 경우가 많아서, LRU가 최선의 정책으로 여겨지지 않는 것도 마찬가지입니다.

테스트 케이스를 하면서 발견한 바로는, 만약 버퍼 캐시 사이즈가 100이고, 데이터는 102개의 데이터를 순차적으로 반복해서 읽는다면 FIFO랑 똑같이 구현되고 cache hit ratio는 0일 것입니다. 이 경우에는 오히려 랜덤 교체정책이나 앞으로 읽힐 데이터들을 미리 버퍼 캐시에 넣어두는 정책이 더 좋을 것입니다. 상황에 맞는 자료구조를 구현 전에 생각해야 되는 것처럼, 내가 구현하는 애플리케이션, 상황에 맞는 교체정책을 사용하는 것이 좋을 것입니다. 모든 상황에 다 최선인 정책은 없고, 모든 정책마다 최악의 상황이 존재한다는 것을 알 수 있습니다.

감사합니다.

[결과 보고에 대한 출처는 학교에서 배운 내용입니다.]