공지 Static Inner Class 2017. 10. 13.

alogrithms 96개의 글 목록닫기

글 제목	조회수	작성일
Expert 공략집 (비공개)	0	2021. 1. 30.
compression 유형 (비공개)	0	2021. 1. 30.
최소 신장 트리 (greedy) (비공개)	0	2020. 2. 1.
B_AdmistrativeDifficults (BST, HashT)	14	2019. 11. 8.
그래프 탐색 비교(dfs, bfs, a-star, dijkstra)	169	2018. 11. 8.
음식배틀 (비공개)	0	2018. 7. 28.
심민구_A_타일채우기 (수정) (검색비허용)	21	2018. 7. 24.
Greedy (Single Source Shortest Path) (비공개)	3	2018. 7. 24.
정기빈_D_연구실 (검색비허용)	2	2018. 7. 24.
정기빈(김민규 수정)_C_루빅의 사각형 <micro-opt> (검색비허용)	2	2018. 7. 24.
이진탐색트리(BST)	44	2018. 7. 24.
Windows 힙 메모리 동작 원리	21	2018. 7. 24.
Memory Pool 최종 종합 포스팅	12	2018. 7. 24.
counting 정렬	4	2018. 7. 24.
Direct Graph Traverse	7	2018. 7. 21.
퀵 정렬	6	2018. 7. 21.
PRO 에 필요한 자료구조/ 알고리즘 (비공개)	1	2018. 7. 4.
2의 보수	2	2018. 6. 2.
[해시테이블]듣보잡 (검색비허용)	3	2018. 5. 25.
mod연산/ 음수 mod 연산 시 문제	749	2018. 5. 6.
Selection 정렬 불안정성	9	2018. 5. 6.
[bfs]째로탈출2 (검색비허용)	3	2018. 4. 30.
[bfs]저글링 죽이기 (검색비허용)	1	2018. 4. 30.
08. 동적계획법	161	2018. 3. 8.
[dp]제곱수의 합 (검색비허용)	9	2018. 3. 3.
[dp] 파도수열 (검색비허용)	2	2018. 3. 1.
[dp] 연속합 (검색비허용)	3	2018. 2. 26.
[dp] 계단 오르기 (검색비허용)	6	2018. 2. 11.
가장 긴 증가하는 부분 수열 :: 풀이 비교 (검색비허용)	4	2018. 1. 26.
[dp] 가장 긴 증가하는 부분 수열 (검색비허용)	3	2018. 1. 26.

글관리 열기 30줄 보기

1 2 3 4

adv algorithms

Memory Pool 최종 종합 포스팅

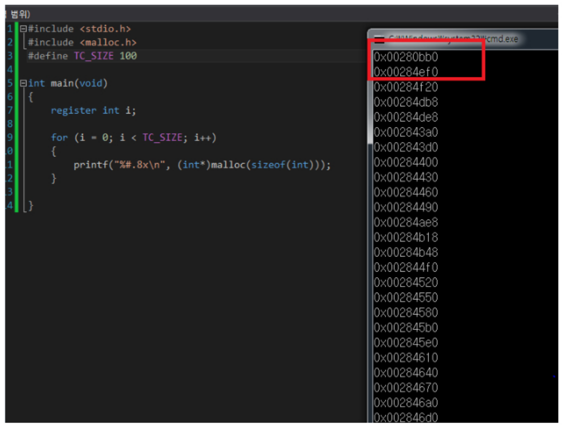
voyager759 · 2018. 7. 24. 20:00

이 포스팅의 내용을 요약하면 다음과 같습니다.

```
메모리 접근은 약 150 cycle 정도가 소요되기 때문에, cache miss rate은 매우 중요하다. logN 복잡도를 가지는 자료구조
(알고리즘)를 사용하더라도, cache miss rate이 최악이면 선형 자료구조에 비해 더 안 좋은 성능을 내기도 한다.

특히 Tree 구조에서 각 노드를 동적할당(malloc)하여 사용하는 경우 cache miss rate 성능이 매우 떨어지기 때문에, '정적
메모리'를 계산하여 사용하거나 '메모리 풀 기법'을 사용하는 것이 좋다.

포스팅 주요 내용>

1) 정적 메모리 Pool
2) 동적 메모리 Pool
```

메모리 풀이 필요한 이유

다음과 같은 실험을 해봤습니다. 100회 4B 동적할당을 하고 그 주소를 출력한 결과입니다. 결과를 보시면 알겠지만, malloc 함수는 연속한 메모리 주소를 반환하지 않습니다. 만약에 우리가 2만개의 데이터로 트리를 형성하고 각 노드를 malloc함수를 통해 동적할당을 한다면 cache miss rate 면에서 매우 저조한 성능을 보일 것 입니다.



이외에도 malloc 함수는 'thread_safety' 등 여러 오버헤드 문제를 가지고 있습니다. 이러한 문제 때문에 특히 알고리즘 문제에서 동적할당은 지양하는 것이 좋습니다. 가장 좋은 방법은 사용될 메모리를 계산하여, 정적 메모리를 pool로 사용하는 것 입니다. 처음에는 구현에 어려움이 있을 수 있으나 다양하게 응용할 경우 쉽게 구현할 수 있는 방법이 있으며, 일일이 free를 해주지 않아도 된다는 장점이 있습니다.

Memory 풀 (정적 , No recycle)

TC 마다 pool 구조체의 ptr을 초기화하면 같은 메모리를 재활용하여 사용하기 때문에, 메모리 측면에서도 효율적입니다.

```
#define FRAG_SIZE 20000
#include<stdio.h>

struct Pool
{
    char buf[FRAG_SIZE];
    char * ptr;
};

void initPool(Pool* pool){

    pool->ptr = pool->buf;
}

void* palloc(Pool* pool, int size){

    void * retAddr = pool->ptr;
    pool->ptr += size;
    return retAddr;
}

Pool pool;

/* 사용 예 */
int main(void)
{
    int tc, TC = 10;
    int w = 0;
    for (tc = 1; tc <= TC; tc++){
        initPool(&pool);
        int * iptr = (int*)palloc(&pool,4*w);
        int * iarr = (int*)palloc(&pool, 4 * 10000 * w); // 10000개 int 배열
        double * dptr = (double*)palloc(&pool, 8*w);
        char * str = (char*)palloc(&pool, 1 * 10000*w); // 10000개 문자열
        printf("pool.ptr : %#.8x\n", pool.ptr);
        w++;
    }

    return 0;
}
```

그렇지만 위와 같은 방식은 메모리 양을 계산할 수 없는 경우 사용할 수 없습니다. 이러한 경우에는 동적 할당을 하되 한 번에 많은 동적 메모리를 할당하는 방식으로 문제를 해결할 수 있습니다. 이러한 방식을 사용할 경우 cache miss rate는 매우 많이 개선될 뿐 아니라, 메모리 확장성 문제도 해결할 수 있습니다.

Memory 풀 (정적 , Recycle)

TC 마다 pool 구조체의 ptr을 초기화하면 같은 메모리를 재활용하여 사용하기 때문에, 메모리 측면에서도 효율적입니다.

```
#define POOL_SIZE 128
typedef int Content;

struct Pool
{
```

```
{
    struct AllocUnit
    {
        AllocUnit* next;
        Content content;
    } pool[POOL_SIZE];
    AllocUnit* recycleBin;
    int availableCount;

public:
    Pool() : availableCount(POOL_SIZE), recycleBin(NULL)
    {
        for (register int i = 0 ; i < POOL_SIZE; i++)
        {
            pool[i].next = NULL;
        }
    }

    Content* palloc()
    {
        Content* temp;
        if (recycleBin)
        {
            return __binAlloc();
        }
        else
        {
            return __newAlloc();
        }
    }

    VOID pfree(Content* ptr)
    {
         AllocUnit* allocUnit = (AllocUnit*)((char*)ptr) - sizeof(AllocUnit*));
         allocUnit->next = recycleBin;
         recycleBin = allocUnit;
    }

private:
    INLINE Content* __binAlloc()
    {
        Content* temp = &recycleBin->content;
        recycleBin = recycleBin->next;
        return temp;
    }

    INLINE Content* __newAlloc()
    {
        if(!availableCount)
        {
            return NULL;
        }
        int index = POOL_SIZE - availableCount;
        --availableCount;
        assert(availableCount >= 0);
        return &pool[index].content;
    }
}
```

Memory 풀 (동적 , No recycle)

이 구현은 recycle을 하지 않기 때문에, 구현이 비교적 간단합니다

```
#include "Pool.h"

int main(void){
    Pool pool;
    int tc, TC = 10;

    for (tc = 1; tc <= TC; tc++){
        pool.init();
        int * iptr = (int*)pool.palloc(4);
        int * iarr = (int*)pool.palloc(4 * 10000); // 10000개 int 배열
        double * dptr = (double*)pool.palloc(8);
        char * str = (char*)pool.palloc(1 * 10000); // 10000개 문자열
    }
    // main 벗어날 경우 자동으로 소멸자 호출
}
```

```
class Pool;
class Frag {

    class Frag * next;
    char buf[FRAG_SIZE];
    char * ptr;
public:
    Frag() : next(nullptr), ptr(buf) {}
    void init() { ptr = buf; }
    friend Pool;
};

class Pool {
    Frag * head;
public:
    Pool() { head = new Frag; }
    void init() {
        Frag* cur;

        for (cur = head; cur; cur = cur->next) {
            cur->init();
        }
    }

    void* palloc(int size) {
        void * retAddr;
        if (head->ptr + size >= head->buf + FRAG_SIZE) {
            Frag* newFrag = new Frag();
            newFrag->next = head;
            head = newFrag;
        }
        retAddr = (void*)head->ptr;
        head->ptr += size;
        return retAddr;
    }
```

```
    ~Pool() {
        Frag* del;

        while (head) {
            del = head;
            head = head->next;
            delete del;
        }
    }
};
```

더 효율적인 방법은 recycle chain을 이용하는 것 입니다. 다음은 민짜고가 구현한 메모리 풀입니다.

Memory 풀 (동적 , Recycle)

```
#include <stdio.h>
#include <malloc.h>
#include <Windows.h>

#define ALLOCATOR_UNIT 100000
#define TINY_TEST      123456
#define SMALL_TEST     1234567
#define LARGE_TEST     12345678

typedef struct item_t {
    int dummy[4];
} item_t;

typedef struct allocunit_t {
    struct allocunit_t* next;
    item_t content;
} allocunit_t;

typedef struct requestunit_t {
    struct requestunit_t* next;
    allocunit_t content[ALLOCATOR_UNIT];
} requestunit_t;

typedef struct {
    int lastRemaining;
    requestunit_t* head;
    allocunit_t* recycleBin;
} allocator_t;

allocator_t allocator = { 0, 0, 0 };

void test(int count, int percent);

item_t* poolAlloc();
void poolFree(item_t* ptr);
void clearPool();
item_t* __binAlloc(allocator_t* allocator);
item_t* __newAlloc(allocator_t* allocator);

int main() {
    test(TINY_TEST, 10);
    test(TINY_TEST, 50);
    test(TINY_TEST, 90);

    test(SMALL_TEST, 10);
    test(SMALL_TEST, 50);
    test(SMALL_TEST, 90);

    test(LARGE_TEST, 10);
    test(LARGE_TEST, 50);
    test(LARGE_TEST, 90);

    return 0;
}

void test(int count, int percent) {
    item_t** array = (item_t**)malloc(sizeof(item_t*)* count);
    int* randArray = (int*)malloc(sizeof(int)* count);
    LARGE_INTEGER start, end, freeend, freq;
    QueryPerformanceFrequency(&freq);
    freq.QuadPart /= 1000;

    for (int i = 0; i < count; ++i)
        randArray[i] = (rand() % 100) < percent;

    printf("********** Item count: %d (%d%% free and reallocate) *********\n", count, percent);
    QueryPerformanceCounter(&start);
    for (int i = 0; i < count; ++i) {
        array[i] = (item_t*)malloc(sizeof(item_t));
    }
    for (int i = 0; i < count; ++i) {
        if (randArray[i]) {
            free(array[i]);
            array[i] = 0;
        }
    }
    for (int i = 0; i < count; ++i) {
        if (!array[i]) {
            array[i] = (item_t*)malloc(sizeof(item_t));
        }
    }
    QueryPerformanceCounter(&end);
    for (int i = 0; i < count; ++i){
        if (array[i]) {
            free(array[i]);
            array[i] = 0;
        }
    }
    QueryPerformanceCounter(&freeend);
    printf("malloc: %llums (+ %llums for cleaning)\n",
        (end.QuadPart - start.QuadPart) / freq.QuadPart,
        (freeend.QuadPart - end.QuadPart) / freq.QuadPart);

    QueryPerformanceCounter(&start);
    for (int i = 0; i < count; ++i) {
        array[i] = poolAlloc();
    }
    for (int i = 0; i < count; ++i) {
```

```
        if (randArray[i]) {
            poolFree(array[i]);
            array[i] = 0;
        }
    }
    for (int i = 0; i < count; ++i) {
        if (!array[i]) {
            array[i] = poolAlloc();
        }
    }
    QueryPerformanceCounter(&end);
    for (int i = 0; i < count; ++i){
        if (array[i]) {
            array[i] = 0;
        }
    }
    clearPool();
    QueryPerformanceCounter(&freeend);
    printf("pool: %llums (+ %llums for cleaning)\n",
        (end.QuadPart - start.QuadPart) / freq.QuadPart,
        (freeend.QuadPart - end.QuadPart) / freq.QuadPart);

    free(array);
    free(randArray);
}

item_t* poolAlloc() {
    if (allocator.recycleBin)
        return __binAlloc(&allocator);
    else
        return __newAlloc(&allocator);
}

void poolFree(item_t* ptr) {
    allocunit_t* allocunit = (allocunit_t*)(((char*)ptr) - sizeof(allocunit_t));
    allocunit->next = allocator.recycleBin;
    allocator.recycleBin = allocunit;
}

void clearPool() {
    requestunit_t* next;
    while (allocator.head) {
        next = allocator.head->next;
        free(allocator.head);
        allocator.head = next;
    }
    allocator.head = 0;
    allocator.lastRemaining = 0;
    allocator.recycleBin = 0;
}

item_t* __binAlloc(allocator_t* allocator) {
    item_t* result = &allocator->recycleBin->content;
    allocator->recycleBin = allocator->recycleBin->next;
    return result;
}

item_t* __newAlloc(allocator_t* allocator) {
    requestunit_t* current;
    int indexToReturn;
    if (!allocator->lastRemaining) {
        allocator->lastRemaining = ALLOCATOR_UNIT;
        current = (requestunit_t*)malloc(sizeof(requestunit_t));
        current->next = allocator->head;
        allocator->head = current;
    }
    indexToReturn = ALLOCATOR_UNIT - allocator->lastRemaining;
    --allocator->lastRemaining;
    return &allocator->head->content[indexToReturn].content;
}
```