개인 제작 라이브러리

작성자 : 송진규

목차

메모리 풀

개요

기본 구조

- 리스트 형태 락프리 스택 구조

구현 목표

- new/delete보다 빠른 속도로 메모리 관리
- Thread safe.
- 메모리 파편 최소화

구현 후 테스트

- **기준 :** new/delete, 메모리 풀 Alloc/Free를 총 1억회 반복

테스트 결과

- new보다 메모리 풀의 Alloc이 약 2배 빠름
- delete보다 메모리 풀의 free가 약 1.5배 빠름

※ 첨부된 파일 MPool_Profiling_1, MPool_Profiling_2 참조

인터페이스_1

```
#define MEMORYPOOL_ENDCODE  890226

template <typename DATA>
class CMemoryPool
{
private:
    /* ********************************************************** */
    // 각 블럭 앞에 사용될 노드 구조체.
    /* ********************************************************** */
    struct st_BLOCK_NODE
    {
        DATA          stData;
        st_BLOCK_NODE *stpNextBlock;
        int           stMyCode;
    };

    /* ********************************************************** */
    // Top으로 사용할 구조체
    /* ********************************************************** */
    struct st_TOP
    {
        st_BLOCK_NODE* m_pTop;
        LONG64 m_l64Count = 0;
    };

private:
    // ---------- 멤버변수 위치를 잡을 때, '캐시 친화 코드(Cache Friendly Code)' 최대한 적용 고려
    // 이 class에서 핵심 함수는 Alloc, Free. 해당 함수의 코드에 맞춰서 멤버변수 배치

    char* m_Memory;                 // 나중에 소멸자에서 한 번에 free하기 위한 변수
    alignas(16) st_TOP m_stpTop;    // Top. 메모리풀은 스택 구조이다.
    int m_iBlockNum;                // 최대 블럭 개수
    bool m_bPlacementNew;           // 플레이스먼트 뉴 여부
    LONG m_iAllocCount;             // 확보된 블럭 개수. 새로운 블럭을 할당할 때 마다 1씩 증가. 해당 메모리풀이 할당한 메모리 블럭 수
    LONG m_iUseCount;               // 유저가 사용 중인 블럭 수. Alloc시 1 증가 / free시 1 감소

public:
    //////////////////////////////////////////////////////////////////////////
    // 생성자
    //
    // Parameters:  (int) 최대 블럭 개수.
    //              (bool) 생성자 호출 여부.
    //////////////////////////////////////////////////////////////////////////
    CMemoryPool(int iBlockNum, bool bPlacementNew = false);

    //////////////////////////////////////////////////////////////////////////
    // 파괴자
```

인터페이스_2

```
    //              (bool) 생성자 호출 여부.
    ////////////////////////////////////////////////////////////////////
    CMemoryPool(int iBlockNum, bool bPlacementNew = false);

    ////////////////////////////////////////////////////////////////////
    // 파괴자
    //
    // 내부에 있는 모든 노드를 동적해제
    ////////////////////////////////////////////////////////////////////
    virtual ~CMemoryPool();


    ////////////////////////////////////////////////////////////////////
    // 블럭 하나를 할당받는다.
    //
    // Parameters: 없음.
    // Return: (DATA *) 데이타 블럭 포인터.
    ////////////////////////////////////////////////////////////////////
    DATA    *Alloc(void);

    ////////////////////////////////////////////////////////////////////
    // 사용중이던 블럭을 해제한다.
    //
    // Parameters: (DATA *) 블럭 포인터.
    // Return: (BOOL) TRUE, FALSE.
    ////////////////////////////////////////////////////////////////////
    bool    Free(DATA *pData);


    ////////////////////////////////////////////////////////////////////
    // 할당된 메모리풀의 총 수.
    //
    // Parameters: 없음.
    // Return: (int) 메모리 풀 내부 전체 개수
    ////////////////////////////////////////////////////////////////////
    int     GetAllocCount(void) { return m_iAllocCount; }

    ////////////////////////////////////////////////////////////////////
    // 사용자가 사용중인 블럭 개수를 얻는다.
    //
    // Parameters: 없음.
    // Return: (int) 사용중인 블럭 개수.
    ////////////////////////////////////////////////////////////////////
    int     GetUseCount(void) { return m_iUseCount; }

};
```

Alloc()함수_1

```
//////////////////////////////////////////////////////////////////////
// 블럭 하나를 할당받는다. (Pop)
//
// Parameters: 없음.
// Return: (DATA *) 데이타 블럭 포인터.
//////////////////////////////////////////////////////////////////////
template <typename DATA>  <T>
DATA*   CMemoryPool<DATA>::Alloc(void)
{
    bool bContinueFlag;

    while (1)
    {
        bContinueFlag = false;

        ////////////////////////////////
        // m_pTop가 NULL일때 처리
        ////////////////////////////////
        if (m_stpTop.m_pTop == nullptr)
        {
            // 새로 만든다.
            st_BLOCK_NODE* pNode = (st_BLOCK_NODE*)malloc(sizeof(st_BLOCK_NODE));
            pNode->stpNextBlock = NULL;
            pNode->stMyCode = MEMORYPOOL_ENDCODE;

            // 플레이스먼트 뉴 호출
            new (&pNode->stData) DATA();

            return &pNode->stData;
        }

        ////////////////////////////////
        // m_pTop가 NULL이 아닐 때 처리
        ////////////////////////////////
        alignas(16)  st_TOP localTop;

        // ---- 락프리 적용 ----
        do
        {
            // 락프리에서 유니크함을 보장하는 값은 Count값.
            // 때문에, 루프 시작 전에 무조건 Count를 먼저 받아온다.
```

Alloc()함수_2

```
        // ---- 락프리 적용 ----
        do
        {
            // 락프리에서 유니크함을 보장하는 값은 Count값.
            // 때문에, 루프 시작 전에 무조건 Count를 먼저 받아온다.
            // top을 먼저 받아올 경우, 컨텍스트 스위칭으로 인해 다른 스레드에서 Count값 변조 후, 변조된 값을 받아올 수도 있다.
            localTop.m_I64Count = m_stpTop.m_I64Count;
            localTop.m_pTop = m_stpTop.m_pTop;

            // null체크
            if (localTop.m_pTop == nullptr)
            {
                // null이라면, 처음부터 다시 로직을 돌린다.
                // flag를 true로 바꾸면, do while문 밖에서 체크 후 continue 실행
                bContinueFlag = true;
                break;
            }

        } while (!InterlockedCompareExchange128((LONG64*)&m_stpTop, localTop.m_I64Count + 1, (LONG64)localTop.m_pTop->stpNextBlock, (LONG64*)&localTop));


        if (bContinueFlag == true)
            continue;

        // 이 아래에서는 모두 스냅샷으로 찍은 localTop을 사용.
        // 아래 작업 중, m_pTop 변주 가능성이 있기 때문에,
        // 플레이스먼트 뉴를 사용한다면 사용자에게 주기전에 '객체 생성자' 호출
        if (m_bPlacementNew == true)
            new (&localTop.m_pTop->stData) DATA();

        return &localTop.m_pTop->stData;
    }
}
```

Free() 함수

```
//////////////////////////////////////////////////////////////////////
// 사용중이던 블럭을 해제한다. (Push)
//
// Parameters: (DATA *) 블럭 포인터.
// Return: (bool) true, false.
//////////////////////////////////////////////////////////////////////
template <typename DATA>
bool    CMemoryPool<DATA>::Free(DATA *pData)
{
    // 이상한 포인터가오면 그냥 리턴
    if (pData == NULL)
        return false;

    // 내가 할당한 블럭이 맞는지 확인
    st_BLOCK_NODE* pNode = (st_BLOCK_NODE*)pData;

    if (pNode->stMyCode != MEMORYPOOL_ENDCODE)
        return false;

    //플레이스먼트 뉴를 사용한다면 메모리 풀에 추가하기 전에 '객체 소멸자' 호출
    if (m_bPlacementNew == true)
        pData->~DATA();

    // ---- 락프리 적용 ----
    st_BLOCK_NODE* localTop;

    do
    {
        // 로컬 Top 셋팅
        // Push는 딱히 카운트가 필요 없다.
        localTop = m_stpTop.m_pTop;

        // 새로 들어온 노드의 Next를 Top으로 찌름
        pNode->stpNextBlock = localTop;

        // Top이동 시도
    } while (InterlockedCompareExchange64((LONG64*)&m_stpTop.m_pTop, (LONG64)pNode, (LONG64)localTop) != (LONG64)localTop);

    return true;
}
```

메모리 풀 TLS

개요

기본 구조

- TLS를 이용한 메모리 풀

구현 목표

- 기존 메모리 풀보다 속도 증가
- 기존 메모리 풀의 기능 계승

구현 스펙

1. Thread 1개(자기 자신)만 접근할 수 있는 공간 필요 **(TLS 사용)**
2. TLS에 일정 수의 Node 보관 **(Chuck 사용. ※ Chuck : Node의 집합)**
3. 각 Thread는 TLS의 Chuck에서 Node를 Alloc한다. Node가 모두 사용되면 다시 Chuck를 가져와, TLS에 보관한다. **(기존에 구현한 메모리 풀 사용. Chuck를 관리)**
4. TLS의 Chuck에 소속된 모든 Node가 사용됐다면, Chuck를 반환한다. **(모든 Node는 1회용)**

개요

구현 후 테스트

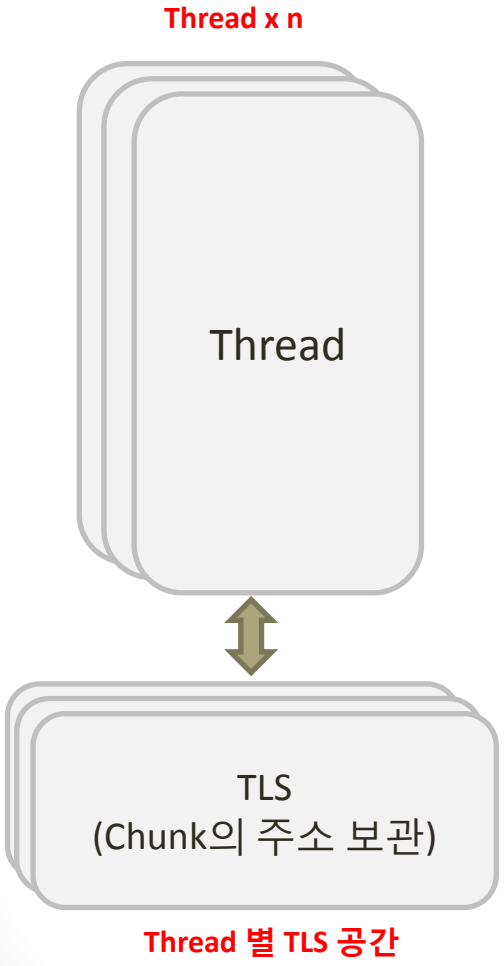
- **기준 :** new/delete, 메모리 풀 Alloc/Free를 총 1억회 반복

테스트 결과

- new보다 약 5배 빠른 Alloc
- delete보다 약 2.5배 빠른 Free

※ 첨부된 파일 MPool_TLS_Profiling_1, MPool_TLS_Profiling_2, MPool_TLS_Profiling_3 참조

내부 구조도



Class CMemoryPoolTLS<DATA>

Class CMemoryPool<stChuck>

메모리 풀
(Chunk를 다룬다)

Struct stChuck

Chuck
(Node의 집합)

Struct stNode

실제 데이터
보관 공간

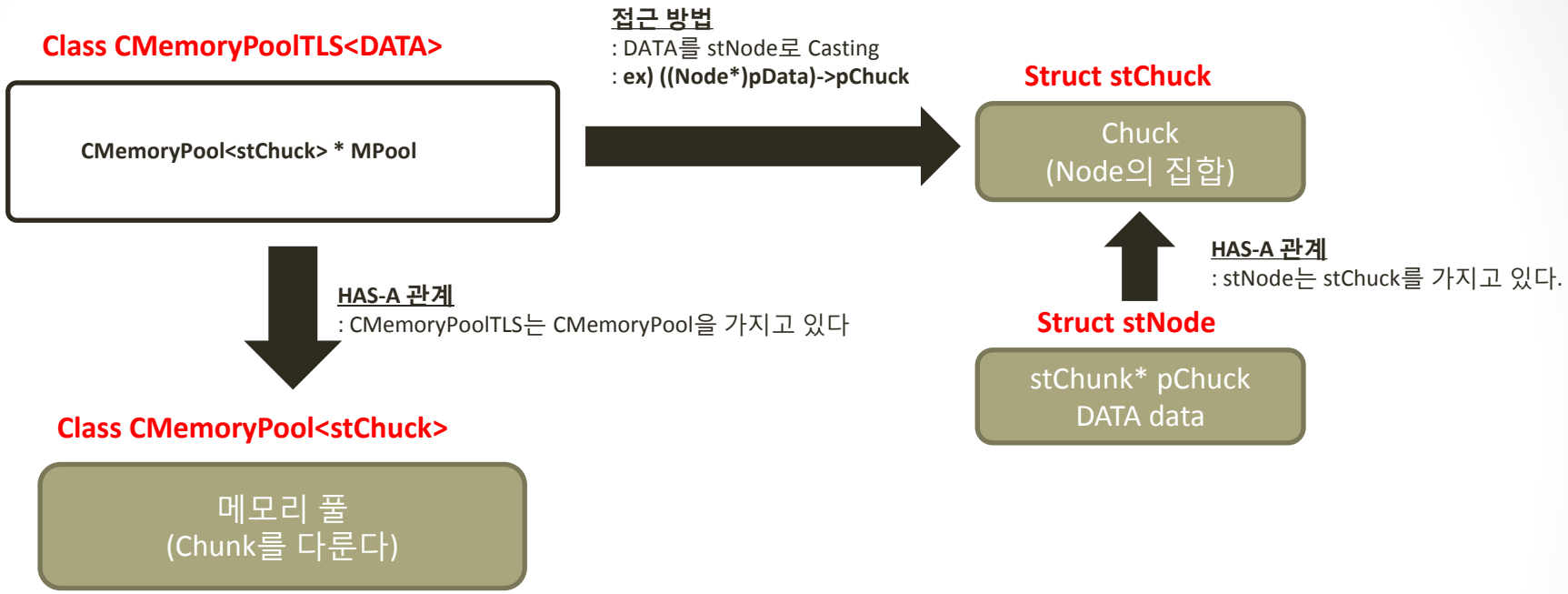
DATA* Alloc()

TLS에 보관중인 Chunk에서 **Node의 data 주소 리턴**

Free(DATA* pData)

pData가 소속된 Chuck의 **모든 Node가 반환**되었다면,
메모리 풀에 Chuck 반환

객체 관계도


Class CMemoryPoolTLS<DATA>
CMemoryPool<stChuck> * MPool
접근 방법
: DATA를 stNode로 Casting
: ex) ((Node*)pData)->pChuck
Struct stChuck
Chuck
(Node의 집합)
HAS-A 관계
: stNode는 stChuck를 가지고 있다.
Struct stNode
stChunk* pChuck
DATA data
HAS-A 관계
: CMemoryPoolTLS는 CMemoryPool을 가지고 있다
Class CMemoryPool<stChuck>
메모리 풀
(Chunk를 다룬다)

인터페이스_1

```
////////////////////////////////////////////////
// 메모리 풀 TLS 버전
//
// 내부에서 청크를 다룸
////////////////////////////////////////////////
namespace Library_Jingyu
{
    template <typename DATA>
    class CMemoryPoolTLS
    {
        struct stChunk;

        // 실제 청크 내에서 관리되는 노드.
        struct Node
        {
            DATA  m_Data;
            stChunk* m_pMyChunk;
            int stMyCode;
        };

    private:
        // ---------------------------
        // 청크 구조체
        // ---------------------------
        struct stChunk
        {

#define NODE_COUNT 300  // 1개의 청크가 다루는 노드의 수

            // 청크 멤버변수

            // ---------- 멤버변수 위치를 잡을 때, '캐시 친화 코드(Cache Friendly Code)' 최대한 적용 고려
            // 이 struct에서는 상위 클래스인  CMemoryPoolTLS의 Alloc, Free가 핵심 함수.
            // 해당 함수의 코드에 맞춰서 멤버변수 배치

            LONG m_iTop;                    // top 겸 Alloc카운트. 0부터 시작
            LONG m_iFreeRef;                // Free 카운트. 0부터 시작
            Node m_arrayNode[NODE_COUNT];   // 실제 다루는 청크 내부 노드. 정적 배열
            CCrashDump* m_ChunkDump;        // 에러 발생 시 덤프를 남길 덤프변수.

            // 청크 생성자
            stChunk();

            // 청크 소멸자
            ~stChunk();
```

인터페이스_2

```
        // 청크 소멸자
        ~stChunk();

        // Alloc카운트 얻기
        int GetAllocCount()
        {
            return m_iTop;
        }

        // Free 카운트 얻기
        LONG GetFreeCount()
        {
            return m_iFreeRef;
        }
    };


private:
    // ------------------
    // 멤버 변수
    // ------------------
    // ---------- 멤버변수 위치를 잡을 때, '캐시 친화 코드(Cache Friendly Code)' 최대한 적용 고려
    // 이 class에서 핵심 함수는 Alloc, Free. 해당 함수의 코드에 맞춰서 멤버변수 배치

    DWORD m_dwIndex;                    // 각 스레드가 사용하는 TLS의 Index
    CMemoryPool<stChunk>* m_ChunkPool;  // TLSPool 내부에서 청크를 다루는 메모리풀 (탁프리 구조)
    static bool m_bPlacementNew;        // 청크 내부에서 다루는 노드 안의 데이터의 플레이스먼트 뉴 호출 여부(청크 아님. 청크 내부안의 노드의 데이터에 대해 플레이스먼트 뉴 호출 여부)
    CCrashDump* m_TLSDump;              // 에러 발생시 덤프를 남길 덤프 변수



public:
    //////////////////////////////////////////////////////////////////////////
    // 생성자
    //
    // Parameters:  (int) 최대 블럭 개수. (청크의 수)
    //              (bool) 생성자 호출 여부. (청크 내부에서 관리되는 DATA들의 생성자 호출 여부. 청크 생성자 호출여부 아님)
    //////////////////////////////////////////////////////////////////////////
    CMemoryPoolTLS(int iBlockNum, bool bPlacementNew);

    //////////////////////////////////////////////////////////////////////////
    // 파괴자
    //
```

인터페이스_3

```
public:
    //////////////////////////////////////////////////////////////////////
    // 생성자
    //
    // Parameters:  (int) 최대 블럭 개수. (청크의 수)
    //              (bool) 생성자 호출 여부. (청크 내부에서 관리되는 DATA들의 생성자 호출 여부. 청크 생성자 호출여부 아님)
    //////////////////////////////////////////////////////////////////////
    CMemoryPoolTLS(int iBlockNum, bool bPlacementNew);

    //////////////////////////////////////////////////////////////////////
    // 파괴자
    //
    // 청크 메모리풀 해제, TLSIndex 반환.
    //////////////////////////////////////////////////////////////////////
    virtual ~CMemoryPoolTLS();

    //////////////////////////////////////////////////////////////////////
    // 블럭 하나를 할당받는다. (Pop)
    //
    // Parameters: 없음.
    // Return: (DATA *) 데이타 블럭 포인터.
    //////////////////////////////////////////////////////////////////////
    DATA* Alloc();

    //////////////////////////////////////////////////////////////////////
    // 사용중이던 블럭을 해제한다. (Push)
    //
    // Parameters: (DATA *) 블럭 포인터.
    // Return: 없음
    //////////////////////////////////////////////////////////////////////
    void Free(DATA* pData);

    // 내부에 있는 청크 수 얻기
    int GetAllocChunkCount()
    {
        return m_ChunkPool->GetAllocCount();
    }

    // 외부에서 사용 중인 청크 수 얻기
    int GetOutChunkCount()
    {
        return m_ChunkPool->GetUseCount();
    }



};
```

Alloc()

```
//////////////////////////////////////////////////////////////////////
// 블럭 하나를 할당받는다. (Pop)
//
// Parameters: 없음.
// Return: (DATA *) 데이타 블럭 포인터.
//////////////////////////////////////////////////////////////////////
template <typename DATA>
DATA* CMemoryPoolTLS<DATA>::Alloc()
{
    // 현재 이 함수를 호출한 스레드의 TLS 인덱스에 청크가 있는지 확인
    stChunk* pChunk = (stChunk*)TlsGetValue(m_dwIndex);

    // TLS에 청크가 없으면 청크 새로 할당함.
    if (pChunk == nullptr)
    {
        pChunk = m_ChunkPool->Alloc();
        if (TlsSetValue(m_dwIndex, pChunk) == FALSE)
        {
            DWORD Error = GetLastError();
            m_TLSDump->Crash();
        }
    }

    // 만약, Top이 NODE_COUNT보다 크거나 같다면 뭔가 잘못된것.
    // 이 전에 이미 캐치되었어야 함
    if (pChunk->m_iTop >= NODE_COUNT)
        pChunk->m_ChunkDump->Crash();

    // 현재 Top의 데이터를 가져온다. 그리고 Top을 1 증가
    DATA* retData = &pChunk->m_arrayNode[pChunk->m_iTop].m_Data;
    pChunk->m_iTop++;

    // 만약, 청크의 데이터를 모두 Alloc했으면, TLS 청크를 NULL로 만든다.
    if (pChunk->m_iTop == NODE_COUNT)
    {
        if (TlsSetValue(m_dwIndex, nullptr) == FALSE)
        {
            DWORD Error = GetLastError();
            pChunk->m_ChunkDump->Crash();
        }
    }

    // 플레이스먼트 뉴 여부에 따라 생성자 호출
    if (m_bPlacementNew == true)
        new (retData) DATA();

    return retData;
}
```

Free()

```
//////////////////////////////////////////////////////////////////////
// 사용중이던 블럭을 해제한다. (Push)
//
// Parameters: (DATA *) 블럭 포인터.
// Return: 없음
//////////////////////////////////////////////////////////////////////
template <typename DATA>
void CMemoryPoolTLS<DATA>::Free(DATA* pData)
{
    // 청크 알아옴
    stChunk* pChunk = ((Node*)pData)->m_pMyChunk;

    // 안전성 검사 -------
    // 이상한 포인터가 오면 그냥 리턴
    if (pData == NULL)
        m_TLSDump->Crash();

    // 내가 할당한 블럭이 맞는지 확인
    if (((Node*)pData)->stMyCode != MEMORYPOOL_ENDCODE)
        m_TLSDump->Crash();

    // FreeRefCount 1 증가
    // 만약 NODE_COUNT가 되면 청크 내부 내용 초기화 후 청크 관리 메모리풀로 Free
    if (InterlockedIncrement(&pChunk->m_iFreeRef) == NODE_COUNT)
    {
        // Free하기전에, 플레이스먼트 뉴를 사용한다면 모든 DATA의 소멸자 호출
        if (m_bPlacementNew == true)
        {
            for (int i = 0; i < NODE_COUNT; ++i)
            {
                pChunk->m_arrayNode[i].m_Data.~DATA();
            }
        }

        // Top과 RefCount 초기화
        pChunk->m_iTop = 0;
        pChunk->m_iFreeRef = 0;

        // 청크 관리 메모리풀로 청크 Free
        if (m_ChunkPool->Free(pChunk) == false)
            m_TLSDump->Crash();
    }
}
```

LockFree - Queue

개요

기본 구조

- Node를 이용한 리스트 기반의 큐

구현 목표

- LockFree 구조로 Thread safe 자료구조 작성
- 락 기반 자료구조보다 빠른 속도

구현 스펙

1. Node를 이용한 리스트 기반의 큐

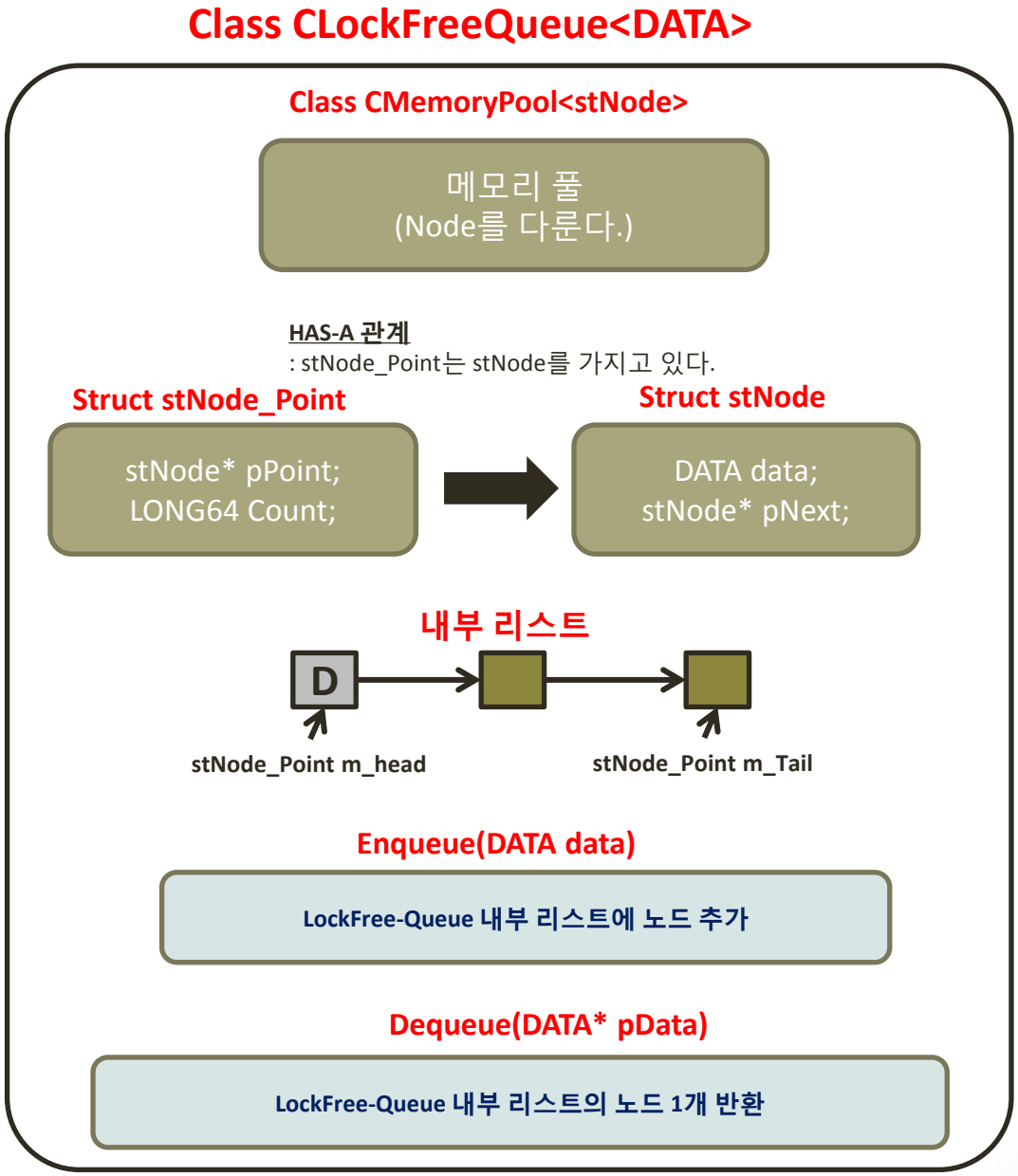
2. 메모리 풀 사용.
→락프리 큐 구조 특성상, 이미 반환된 메모리에 접근하는 경우가 있기 때문에, 전용 메모리 풀 필요

3. Double Compare and set와 유니크한 Count값을 이용
→ ABA문제 방지

4. Head는 무조건 Dummy.

5. 실제 데이터는 head->next부터.
→ head가 null이 되는 상황 방지.

내부 구조도


Class CLockFreeQueue<DATA>
Class CMemoryPool<stNode>
메모리 풀
(Node를 다룬다.)
HAS-A 관계
: stNode_Point는 stNode를 가지고 있다.
Struct stNode_Point
stNode* pPoint;
LONG64 Count;
Struct stNode
DATA data;
stNode* pNext;
내부 리스트
D
stNode_Point m_head
stNode_Point m_Tail
Enqueue(DATA data)
LockFree-Queue 내부 리스트에 노드 추가
Dequeue(DATA* pData)
LockFree-Queue 내부 리스트의 노드 1개 반환

인터페이스_1

```
template <typename T>
class CLF_Queue
{
    // 노드
    struct st_LFQ_NODE
    {
        T m_Data;
        st_LFQ_NODE* m_stpNextBlock;
    };

    // tail용 구조체
    struct st_NODE_POINT
    {
        st_LFQ_NODE* m_pPoint = nullptr;
        LONG64 m_l64Count = 0;
    };

    // 내부의 노드 수
    LONG m_NodeCount;

    // 메모리풀
    CMemoryPool<st_LFQ_NODE>* m_MPool;

    // 시작 노드 가리키는 포인터
    alignas(16) st_NODE_POINT m_stpHead;

    // 마지막 노드 가리키는 포인터
    alignas(16) st_NODE_POINT m_stpTail;

    // 에러났을 때 크래시 내는 용도
    CCrashDump* m_CDump;



public:
    // 생성자
    // 내부 메모리풀의 플레이스먼트 뉴 사용여부 인자로 받음.
    // 디폴트는 false (플레이스먼트 뉴 사용 안함)
    CLF_Queue(int PoolCount, bool bPlacementNew = false);

    // 소멸자
    ~CLF_Queue();

    // 내부 노드 수 얻기
    LONG GetInNode();
```

인터페이스_2

```
    // 내부 노드 수 얻기
    LONG GetInNode();

    // Enqueue
    void Enqueue(T data);

    // Dequeue
    // out 매개변수. 값을 채워준다.
    //
    // return -1 : 큐에 할당된 데이터가 없음.
    // return 0 : 매개변수에 성공적으로 값을 채움.
    int Dequeue(T& OutData);

};
```

Enqueue()

```
// Enqueue
template <typename T>
void CLF_Queue<T>::Enqueue(T data)
{
    st_LFQ_NODE* NewNode = m_MPool->Alloc();

    NewNode->m_Data = data;
    NewNode->m_stpNextBlock = nullptr;

    // 락프리 구조 (카운터 사용) ------------
    alignas(16) st_NODE_POINT LocalTail;

    while (true)
    {
        // tail 스냅샷
        LocalTail.m_l64Count = m_stpTail.m_l64Count;
        LocalTail.m_pPoint = m_stpTail.m_pPoint;

        // LocalTail의 Next 스냅샷
        st_LFQ_NODE* LocalNext = LocalTail.m_pPoint->m_stpNextBlock;

        // 정말 m_stpTail이 LocalTail와 같다면 로직 진행
        if (LocalTail.m_l64Count == m_stpTail.m_l64Count)
        {
            // Next가 Null이면 로직 진행
            if (LocalNext == nullptr)
            {
                // 라인 연결 시도
                if (InterlockedCompareExchange64((LONG64*)&m_stpTail.m_pPoint->m_stpNextBlock, (LONG64)NewNode, (LONG64)nullptr) == (LONG64)nullptr)
                {
                    // 라인이 연결되면 Tail이동 시도
                    InterlockedCompareExchange128((LONG64*)&m_stpTail, LocalTail.m_l64Count + 1, (LONG64)NewNode, (LONG64*)&LocalTail);
                    break;
                }
            }

            else
            {
                // null이 아니라면 Tail이동작업 시도
                InterlockedCompareExchange128((LONG64*)&m_stpTail, LocalTail.m_l64Count + 1, (LONG64)LocalNext, (LONG64*)&LocalTail);
            }
        }

    }

    // 큐 내부의 사이즈 1 증가
    InterlockedIncrement(&m_NodeCount);
}
```

Dequeue() - 1

```
// Dequeue
// out 매개변수. 값을 채워준다.
//
// return -1 : 큐에 할당된 데이터가 없음.
// return 0 : 매개변수에 성공적으로 값을 채움.
template <typename T>
int CLF_Queue<T>::Dequeue(T& OutData)
{
    // 큐 내부의 사이즈 1 감소
    // 노드가 없으면 1 올리고 리턴
    // !! 동시에 2개의 스레드가 사이즈가 1인 상태에서 디큐를 시도하면 문제가 생김 !!
    // !! 먼저 여길 통과한 스레드가 0으로 만들고, 다른 스레드가 -1로 만들어 버림 !!
    // !! 즉, 정상적인 상황인데 노드 카운트가 -1이 됨. !!
    // !! 노드 수가 -1이면, 다시 1 올려서 0으로 만들고 나간다. !!
    if (InterlockedDecrement(&m_NodeCount) < 0)
    {
        InterlockedIncrement(&m_NodeCount);
        return 0;
    }

    // 락프리 구조 ------------------------
    // 디큐는, 기본적으로 현재 헤드가 가리키고 있는 노드의, 다음 노드의 값을 리턴한다.
    // 헤드는 무조건 더미를 가리키는 것이다.
    alignas(16) st_NODE_POINT LocalHead, LocalTail;

    while (true)
    {
        // head 스냅샷
        LocalHead.m_I64Count = m_stpHead.m_I64Count;
        LocalHead.m_pPoint = m_stpHead.m_pPoint;

        // tail 스냅샷
        LocalTail.m_I64Count = m_stpTail.m_I64Count;
        LocalTail.m_pPoint = m_stpTail.m_pPoint;

        // LocalHead의 Next 스냅샷
        st_LFQ_NODE* LocalHead_Next = LocalHead.m_pPoint->m_stpNextBlock;
```

Dequeue() - 2

```
        // tail 스냅샷
        LocalTail.m_I64Count = m_stpTail.m_I64Count;
        LocalTail.m_pPoint = m_stpTail.m_pPoint;

        // LocalHead의 Next 스냅샷
        st_LFQ_NODE* LocalHead_Next = LocalHead.m_pPoint->m_stpNextBlock;

        // 정말 m_stpHead이 LocalHead와 같다면 로직 진행
        if (m_stpHead.m_I64Count == LocalHead.m_I64Count)
        {
            // 헤더와 테일이 같으면서
            if (LocalHead.m_pPoint == LocalTail.m_pPoint)
            {
                // LocalHead->Next가 null인지 체크
                if (LocalHead_Next == nullptr)
                    continue;

                // 그게 아니면 tail 이동 시도
                else
                    InterlockedCompareExchange128((LONG64*)&m_stpTail, LocalTail.m_I64Count + 1, (LONG64)LocalTail.m_pPoint->m_stpNextBlock, (LONG64*)&LocalTail);
            }

            // 헤더와 테일이 다르면 디큐작업 진행
            else
            {
                // LocalHead->Next가 null인지 체크
                if (LocalHead_Next == nullptr)
                    continue;

                // 리턴할 데이터 받아두기 ------
                OutData = LocalHead_Next->m_Data;

                // 헤더 이동 시도
                if (InterlockedCompareExchange128((LONG64*)&m_stpHead, LocalHead.m_I64Count + 1, (LONG64)LocalHead_Next, (LONG64*)&LocalHead))
                {
                    // 이동 전의 헤더를 메모리풀에 반환
                    m_MPool->Free(LocalHead.m_pPoint);
                    break;
                }
            }
        }
    }

    // 성공했으니 0 리턴
    return 0;
}
```

LockFree - Stack

개요

기본 구조

- Node를 이용한 리스트 기반의 큐

구현 목표

- LockFree 구조로 Thread safe 자료구조 작성
- 락 기반 자료구조보다 빠른 속도

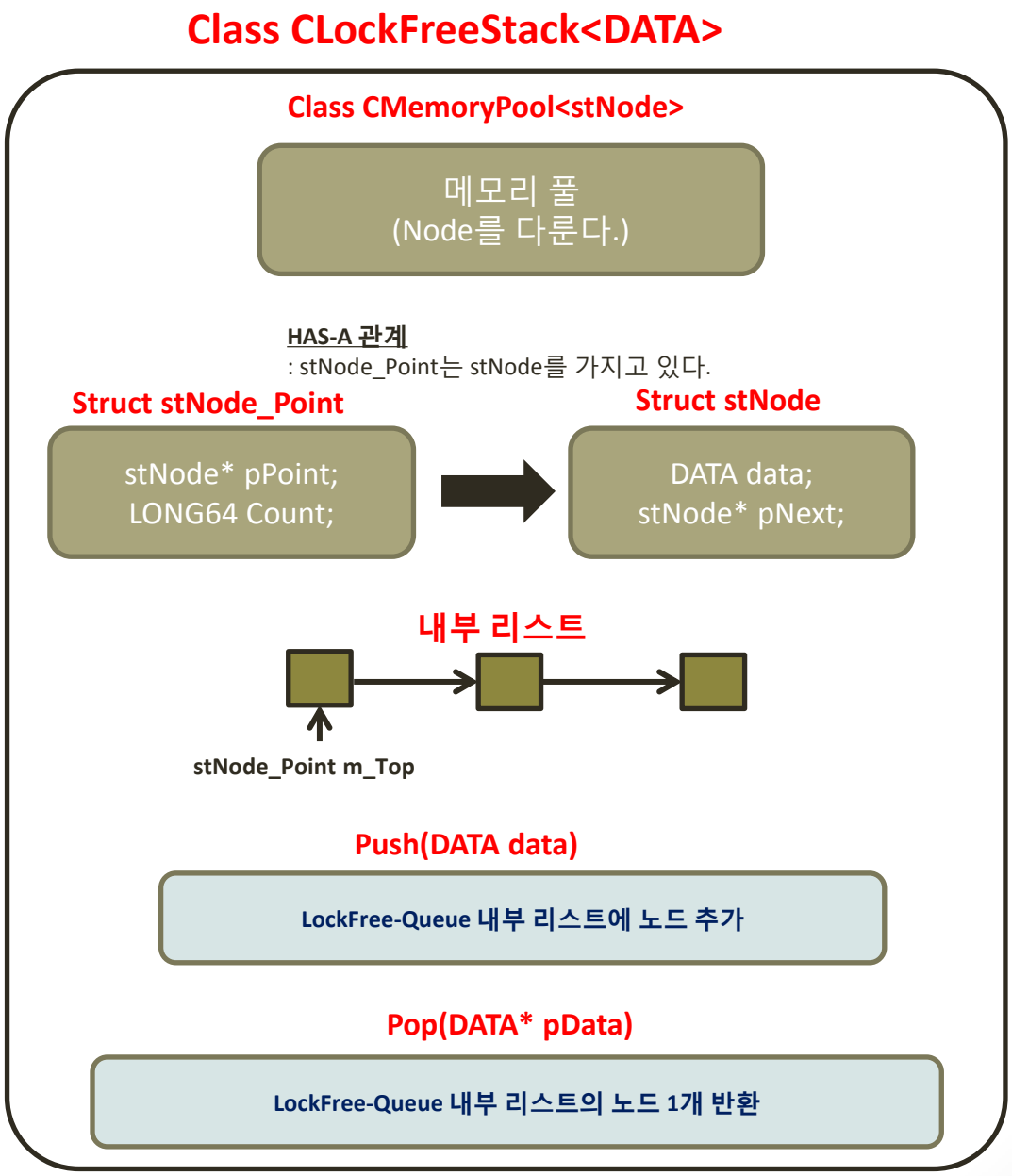
구현 스펙

1. Node를 이용한 리스트 기반의 스택

2. 메모리 풀 사용.
→LockFree 구조 특성상, 이미 반환된 메모리에 접근하는 경우가 있기 때문에, 전용 메모리 풀 필요

3. Double Compare and set와 고유한 Count값을 이용
→ ABA문제 방지

내부 구조도


Class CLockFreeStack<DATA>
Class CMemoryPool<stNode>
메모리 풀
(Node를 다룬다.)
HAS-A 관계
: stNode_Point는 stNode를 가지고 있다.
Struct stNode_Point
stNode* pPoint;
LONG64 Count;
Struct stNode
DATA data;
stNode* pNext;
내부 리스트
stNode_Point m_Top
Push(DATA data)
LockFree-Queue 내부 리스트에 노드 추가
Pop(DATA* pData)
LockFree-Queue 내부 리스트의 노드 1개 반환

인터페이스_1

```
template <typename T>  <T>
class CLF_Stack
{
private:

    // 노드
    struct st_LFS_NODE
    {
        T m_Data;
        st_LFS_NODE* m_stpNextBlock;
    };

    // Top 구조체
    struct st_TOP
    {
        st_LFS_NODE* m_pTop = nullptr;
        LONG64 m_l64Count = 0;
    };

    // ---------- 멤버변수 위치를 잡을 때 '캐시 친화 코드(Cache Friendly Code)' 최대한 적용 고려
    // 이 class에서 핵심 함수는 Push, Pop. 해당 함수의 코드에 맞춰서 멤버변수 배치

    // 메모리풀
    CMemoryPool<st_LFS_NODE>* m_MPool;

    // Top을 가리키는 구조체 변수
    alignas(16) st_TOP m_stpTop;

    // 리스트 내부의 노드 수
    LONG m_NodeCount;

    // 에러났을 때 크래시 내는 용도
    CCrashDump* m_CDump;



public:
    // 생성자
    // 내부 메모리풀의 플레이스먼트 뉴 사용여부 인자로 받음.
    // 디폴트는 false (플레이스먼트 뉴 사용 안함)
    CLF_Stack(bool bPlacementNew = false);

    // 소멸자
    ~CLF_Stack();
```

인터페이스_2

```
    // 소멸자
    ~CLF_Stack();

    // 내부 노드 수 얻기
    LONG GetInNode();

    // 인덱스를 스택에 추가
    //
    // return : 없음 (void)
    void Push(T Data);

    // 인덱스 얻기
    //
    // 성공 시, T 리턴
    // 실패시 Crash 발생
    T Pop();
};
```

Push()

```
template <typename T>
void CLF_Stack<T>::Push(T Data)
{
    // 새로운 노드 할당받은 후, 데이터 셋팅
    st_LFS_NODE* NewNode = m_MPool->Alloc();
    NewNode->m_Data = Data;

    // ---- 락프리 적용 ----
    st_LFS_NODE* localTop;

    do
    {
        // 로컬 Top 셋팅
        localTop = m_stpTop.m_pTop;

        // 신 노드의 Next를 Top으로 설정
        NewNode->m_stpNextBlock = localTop;

        // Top이동 시도
    } while (InterlockedCompareExchange64((LONG64*)&m_stpTop.m_pTop, (LONG64)NewNode, (LONG64)localTop) != (LONG64)localTop);

    // 내부 노드 수 증가.
    InterlockedIncrement(&m_NodeCount);
}
```

Pop()

```
template <typename T>
T CLF_Stack<T>::Pop()
{
    // 더 이상 pop 할게 없으면 Crash 발생
    if (m_stpTop.m_pTop == nullptr)
        m_CDump->Crash();

    // 내부 노드 수 감소
    InterlockedDecrement(&m_NodeCount);

    // ---- 락프리 적용 ----
    alignas(16)  st_TOP localTop;

    do
    {
        localTop.m_I64Count = m_stpTop.m_I64Count;
        localTop.m_pTop = m_stpTop.m_pTop;

        // null체크
        if (localTop.m_pTop == nullptr)
        {
            m_CDump->Crash();
        }

    } while (!InterlockedCompareExchange128((LONG64*)&m_stpTop, localTop.m_I64Count + 1, (LONG64)localTop.m_pTop->m_stpNextBlock, (LONG64*)&localTop));

    // 리턴할 데이터 받아두기
    T retval = localTop.m_pTop->m_Data;

    // pop 노드 Free하기
    m_MPool->Free(localTop.m_pTop);

    // 리턴
    return retval;
}
```