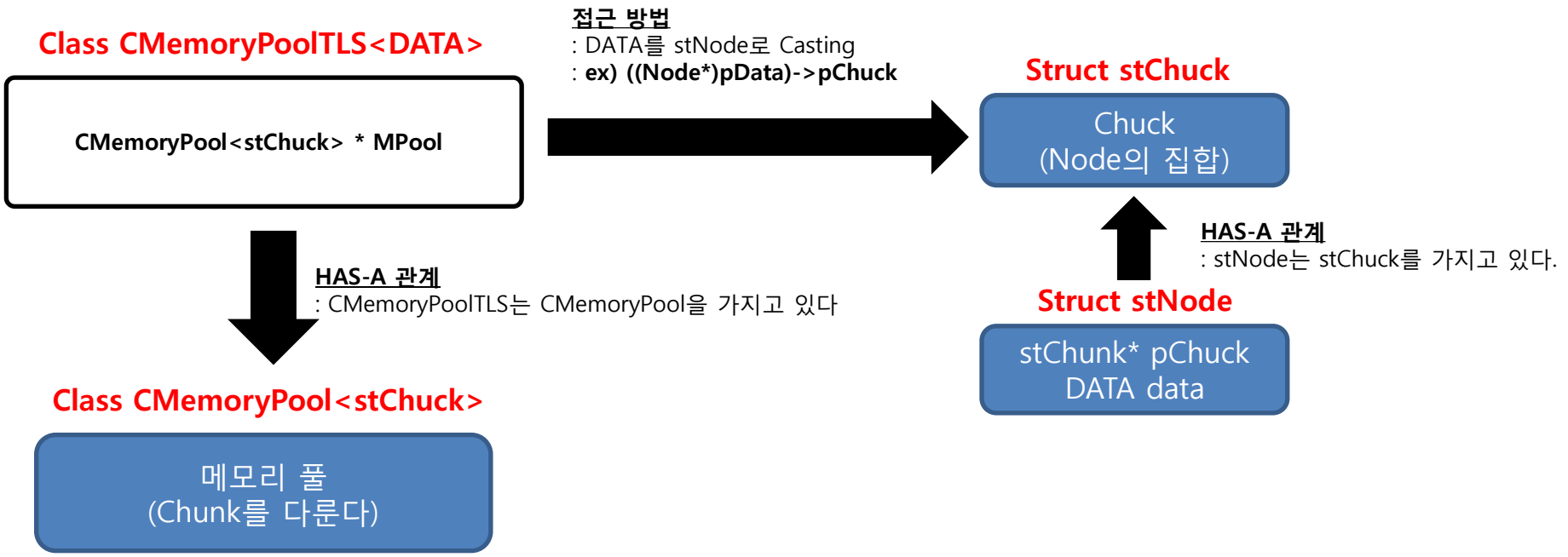
목표

1. New, delete보다 빠른 동적 할당, 동적 해제
2. Thread Safe 구조
3. 기존 메모리 풀보다 속도 향상

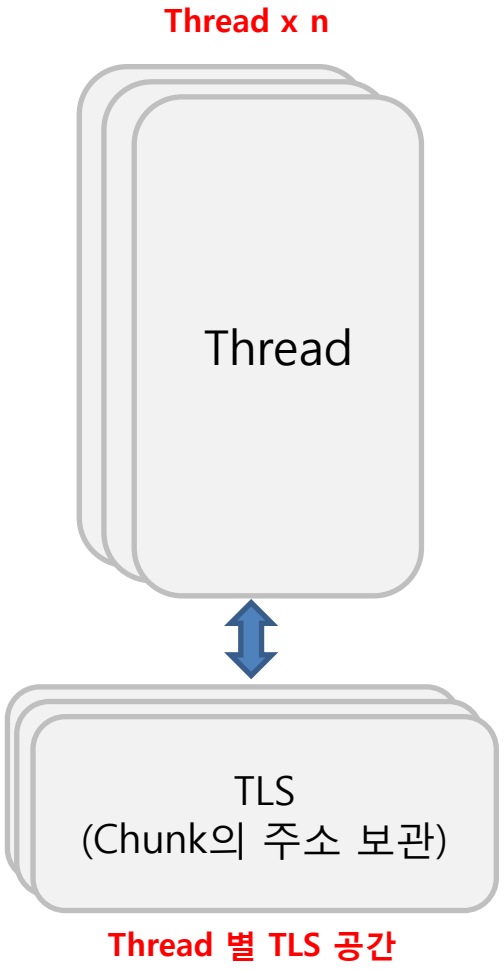
구현 스펙

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

메모리 풀 TLS의 객체 관계도



메모리 풀 TLS 기본 구조



Class CMemoryPoolTLS<DATA>

Class CMemoryPool<stChuck> * MPool

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

예시) Alloc

Class CMemoryPoolTLS<DATA>

CMemoryPool<stChuck>* MPool

메모리 풀
(Chunk를 다룬다)

Thread x n

Thread

Alloc() 함수 호출

DATA* Alloc()

2. 없다면, 메모리 풀에서 Chunk를 얻어온다.

4. 유저에게 Node의 data 주소 리턴

```
// 1. 이 함수를 호출한 스레드의 TLS에 Chuck가 있는지 확인
stChuck* pChuck = TlsSetValue(Index);

// Chuck가 없으면 메모리 풀에서 새로 할당
If(pChuck == nullptr)
{
    // 2. 메모리 풀에서 Chuck를 Alloc
    pChuck  = Mpool.Alloc();

    // 3. Alloc한 Chuck를 TLS에 보관
    TlsSetValue(Index, pChuck );
}

// 리턴할 데이터의 주소 받아두기
DATA* Ret = &(pChuck->alloc가능한 Node->Data)

// 모든 데이터가 Alloc되었다면 TLS를 nullptr로 만든다
If(모든 데이터 Alloc)
{
    TlsSetValue(Index, nullptr);
}

// 4. 데이터의 주소 리턴.
Return Ret;
```

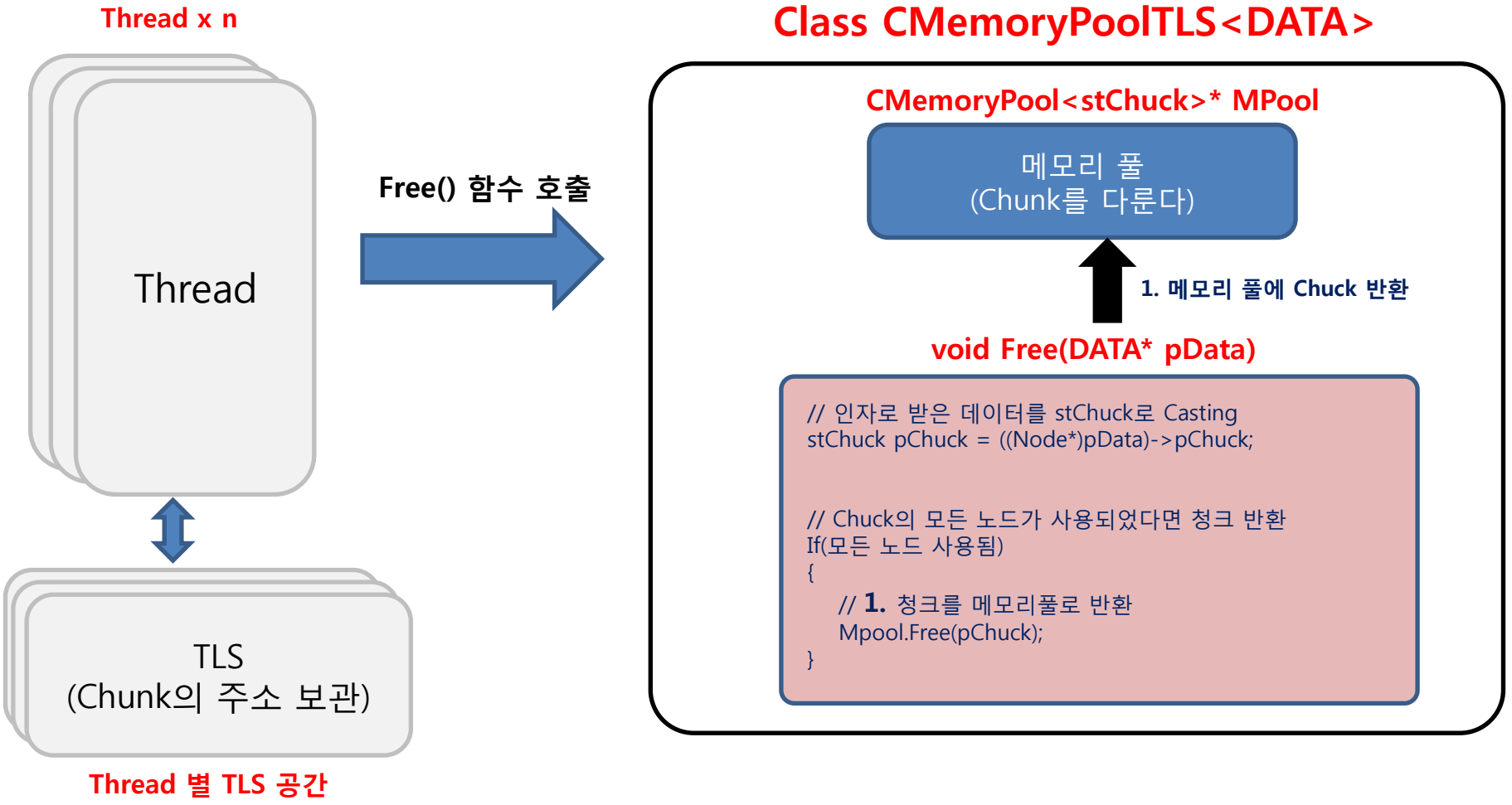
1. 동적 TLS 공간에 Chunk가 있는지 확인

TLS
(Chunk의 주소 보관)

Thread 별 TLS 공간

3. 얻어온 Chuck의 주소를 TLS에 보관

예시) Free


Thread x n
Thread
TLS
(Chunk의 주소 보관)
Thread 별 TLS 공간
Free() 함수 호출
Class CMemoryPoolTLS<DATA>
CMemoryPool<stChuck>* MPool
메모리 풀
(Chunk를 다룬다)
1. 메모리 풀에 Chuck 반환
void Free(DATA* pData)
// 인자로 받은 데이터를 stChuck로 Casting
stChuck pChuck = ((Node*)pData)->pChuck;
// Chuck의 모든 노드가 사용되었다면 청크 반환
If(모든 노드 사용됨)
{
// 1. 청크를 메모리풀로 반환
Mpool.Free(pChuck);
}

구현 후 테스트

1. new, delete, 메모리 풀 Alloc, 메모리 풀 Free를 1억회 테스트 진행

결과

1. New보다 약 5배 빠른 Alloc 구현
2. Delete보다 약 2.5배 빠른 Free 구현

※ 첨부된 파일 Profiling_1 ~ Profiling_3 참조