**<8.10>**

**SpinLock (복습)**

싱글코어에서 스핀락이 돌아가는경우, 아무런 의미가없음.

**당연히 멀티코어이상, 듀얼CPU이상인 경우에 한해서 사용해야 한다.**

**내 퀀텀타임 안에 상대방의 락이 풀릴 가능성이 있다면 가장 좋을 것이고,**

**SpinLock을 도는동안에는 CPU를 100%점유하고 갈것이다.**

따라서 아주 짧은 시간에 풀릴 가능성이 있는것들로 시도해야 한다.

**SpinLock**

**SpinLock은 내 퀀텀타임 내에 자원을 획득할 수있다면, 가장 이상적일 것이다.(성능향상)**

**만약 퀀텀타임을 모두 소모하면서 돌게되면, 다른 스레드가 돌 기회를 잃어버리기 때문에 동기화객체를 사용하는것보다 (전체적인) 성능이 떨어질것이다.**

SpinLock역시 퀀텀타임을 모두 소모한다면 컨텍스트 스위칭이 일어난다.

퀀텀타임이 20m/s인데, SpinLock을 걸려고하는 지점이 19m/s일수도 있음.

따라서 퀀텀타임 내에 진입하기를 희망하는 것일뿐, 이를 보장받지는 못한다.

요즘 스핀락에는 하이퍼스레딩이 지원되는 환경에서 yeildProcesser를 통해 한번씩 다른스레드가 돌 기회를 제공한다.

이는 컨텍스트 스위칭이 아니며, OS입장에서는 하나의 코어가 자기할일을 열심히 하는것으로 인식한다.

**동기화 객체**

**동기화객체는 1m/s안에 임계영역을 얻을수 있다고 하더라도 퀀텀타임을 그 즉시 포기하고 스레드가 block 걸리게 되므로, 다시 ReadyQ에 들어가서 스케쥴러에 의해 실행되기까지 오랜시간이 걸리게된다.**

단, SRWLock에서는 스레드를 block걸기전에 SpinLock을 일정회수 돌려보고 들어간다.

SpinLock은 CPU차원에서의 기다림(루프)를 말하는 것으로, SRWLock과는 별개이다.

**Sleep(0)**

Sleep(0)의 경우 스케쥴링 되어 내 ReadyQ의 가장 뒤쪽에 큐잉된다.

동기화객체의 경우 block -> ready -> running되기 때문에 Sleep(0)보다 느리다.

**커널모드 전환**

동기화객체를 사용했고, 독점적으로 접근할수 있는경우라면(경합이 발생하지않은경우) **(X)**

커널모드 객체일 경우 (우리는 커널모드 동기화객체안씀) **(O)**

유저모드 객체일 경우 내가 획득할 수 있다면 **(X)**

(단순히 인터락 비트체크를 하고 들어갈 것.)

요즘 SRWLock은 동기화가 걸려도 완벽한 유저모드로 돌아간다. **(X)**

**피터슨 알고리즘**

**피터슨 알고리즘은 인터락 함수를 쓰지않고 단순히 flag체크하여 두개의 공간을 두고 두개의 스레드가 서로 안전하게 진입하는 것을 목표로 삼은 알고리즘이다.**

이는 완성되지못한 알고리즘이다.

**우리가 동기화객체는 어떤 스레드든 독점적인 접근이 된다는것이지, 스레드 두개만을 넣고 바꿔가며 돌아가는 것이 아니다.**

**SpinLock의 구현**

**SpinLockFlag = false;**

변수를 하나두고,

**InterlockedExchange()**

를 사용해 진입여부를 판단한다.

**InterlockedCompareExchange()는 비교와 교체를 동시에 원자적으로 수행하는것으로,**

**InterlockedExchange()보다 무겁다.**

**yeildProcesser 명령어를 넣어 한번씩 쉬는타임을 줄수도 있겠다.**

(다른스레드에게 기회제공)

추가적으로, 일반 if문을 체크하여 Interlocked의 호출회수를 줄이는것도 가능하다.

**SpinLock이 필요한 경우 CriticalSection내부의 SpinLockCount를 이용하든가, SRWLock에 내내장된 것 사용한다.**

**SpinLock을 구현하여 사용하지않는 이유**

OS커널 내부 같은 경우 실행의요소가 아주 단시간으로 만들어진 요소들이기 때문에 거의대부분 SpinLock으로 이루어져있다.

우리가 이를 컨텐츠에서 구현한다면 역효과가 날 가능성이 너무 크기 때문에 직접사용하지않는다.

**Lock-Free알고리즘**

락프리 알고리즘은 스핀락과 비슷한 부분이 있다.

Lock-Free는 내부적으로 락을 걸지않는다. (busy wating이라는 표현도 존재)

SpinLock은 개념적으로는 Lock이지만 동기화객체를 사용하여 스레드를 block시키는 것이 아니라, 일이 끝날때까지 기다린다.

**SpinLock은 접근을 목적으로 하는 기다림이고,**

**LockFree는 최종적인 행위에 대한 commit을 돌린다.**

**LockFree는 진입의 과정에서 검사하지않고, 일단 진입한다.**

**여러 개의 요청중 하나가 성공했다면 다른것들은 재시도한다**.

**LockFree Stack에 pop을 한다고 할 경우, 최종적으로 pop을 확정짓기 직전에 스택의 변화의 유무를 판단하여 commit하거나 재시도한다.**

**따라서 SpinLock과 Lock-Free는 대기의 목적이 다르다**

(락이 없다고하여 n개의 요청이 동시에 돌아가는 개념이 아니므로, 성능이 빠르다고 말하기에는 애매한 부분이다.)

**Lock-Free알고리즘 예시**

도착점이 있다고 가정하고, 제일먼저 도착하는 사람만 통과한다고 가정한다.

이때 한명을 제외한 나머지는 다시 출발선에서 달려야한다.

**SpinLock의 경우 누군가 출발했다면 출발하지않은 채로 출발선에서부터 기다리는 것이고,**

**Lock-Free는 다같이 출발하되 실패한 사람은 출발선에서부터 다시 달리는 것.**

Lock-Free가 좋은지/나쁜지에 대한 판단은 직접 구현하여 판단해 볼 것.

**Lock-Free알고리즘을 만드는 이유**

커리큘럼상 메모리풀, 링버퍼 등에 Lock-Free 알고리즘을 적용할 것이다.

이는 성능상의 이유가 아닌, 검증차원에서의 활용이다.

왜냐면 락프리큐, 락프리스택은 제대로 구현하기가 쉽지않다.

**Lock-Free Stack/Q의 구현을 증명하기위해서는 모든 문제상황의 시나리오를 설명하고 만들 수 있어야 한다.**

(인터넷상에 논문/자료가 많으나 완벽하게 제대로 구현된 것은 없다고 생각함)

Lock-Free알고리즘을 구현했다고 한다면 실무에서는 100% 믿지않을 것.

심지어 NC, Nexon에 있는 개발자역시도 본인이 만든 것을 믿지못하기 때문에 사용하지않는다.

**사전작업과 최종확인 작업**

예시코드에서는 do while로 들어간다.

push라는 과정에서 사전작업과 최종 확정작업(commit)으로 나뉜다.

**사전작업은 자료구조에 변화를 두지않고, 나혼자 진행하는 작업을 뜻한다.**

**최종확인 작업은 반영하고자 하는 자료구조에 한번에 반영하는 것을 뜻한다.**

**포인터를 교체하는 한번의 행위로 commit이 이뤄져야 하므로, list로만 구현이 가능하다.**

**Lock-Free Stack에서는 Top에 대한 변화유무에 따라 재시도/PUSH가 결정날 것이다.**

동기화객체를 통해 임계영역을 독점적으로 점유한다면 문제될게 없다.

이렇게까지 성능을 신경쓰지 않기때문에, 대부분 동기화를걸고 안전하게 간다.

**CAS(Compare-And-Swap)**

**CAS(Compare-And-Swap)란, 비교와 교체까지 atomic하게 해주는 것을 뜻한다.**

Windows 환경에서는 InterlockedCompareExchange()가 될것이다.

(리눅스 쪽에서도 리눅스만의 함수가 존재할 것.)

**LockFree는 싱글리스트 형태로**

LockFree를 더블리스트로 적용하고자 한다면, 변경되어야 할게 너무 많다.

나를기준으로 앞뒤의 노드를 연결하는 것은 상관없겠지만,

**내 노드를 기준으로 next의 prev와, prev의 next를 atomic하게 동시에 바꿀방법이 없으므로, 구현이 불가능하다.**

**따라서 LockFree Q/Stack은 모두 싱글리스트 형태로 구현한다.**

**((**

**어차피 노드 앞뒤에만 추가/삭제될것인데 중간에 노드를 넣을 걱정을 해야됨??**

**Stack을 더블리스트로 락프리구현한다고 한다면..**

**이전에 저장한 NewNode의 Next를 비교하여, 같다면 Top을 NewNode로 지정함.**

**더블리스트의 경우 NewNode->Next->Prev가 NewNode가 되어야함. 할일추가**

**))**

**LockFree-Stack의 구현**

**1.먼저 기존의 Top을 백업한다.**

Node\* CopyTopNode = TopNode;

**2. 그리고 내 노드를 준비하여, Top에다가 연결한다.**

(이는 내 노드가 생성되어 next로 Top을 가리키는 것으로, 자료구조랑은 상관이없음)

**(Node1 ->) TopNode -> TopNodeNext**

**(Node1은 자료구조 쪽에서 모름)**

**3. 백업한 Top노드가 현재 자료구조의 Top노드와 일치하는 지 비교한다.**

당연히 비교 및 대입은 CAS로(InterlockedCompareExchange()) 진행한다.

**3-1. 백업한 Top노드 현재 자료구조의 Top노드가 같다면?**

If(CopyTopNode == TopNode)

{

TopNode = Node1;

}

**Node1 -> TopNode -> TopNodeNext**

**새노드를 넣고 실질적인 commit.**

**3-2. 백업한 Top노드 != 현재 자료구조 Top노드**

우리가 작업하던사이 새로운 노드(Node2)가 Top을 찌르고 PUSH한 경우,

백업한TOP노드와 일치하지않는다.

**따라서 다시 재시도하러 돌아간다.**

(로컬에 기존의 TOP저장...)

**동시에 치고들어오는 경우 순서**

동시에 치고들어오는 순서는 어떤것도 보장받을 수 없다.

애초에 보장 받을 수 있는 방법이 없으며, 이를 문제삼지 않는다.

**LockFree에서 경합**

일반적인 스택을 기반으로 구현한다면 경합이 많이 발생할 일은 없을 것이다.

**하지만 우리는 결국 락프리스택, 락프리큐, 락프리 메모리풀을 만들어 최종적으로 테스트한다.이때 코어 개수이상으로 스레드를 만들어 push/pop을 반복적으로 한다면 해당 루프가 1,20만번 까지 돌 수 있다.**

이 같은 경우는 어쩔수없음

**ABA 문제**

모든 락프리 자료구조에는 ABA문제가 나타난다.

(CAS가 오작동할 일은 없으므로, 후보에서 제외한다.)

**LOCK-FREE 진행상황으로,**

1. Top을 백업(지역에 보관)

2. 새노드에 top을 연결

3. 백업된 top과 현재 자료구조 top을 비교.

**이때 PUSH의 경우는 (3)항목 체크만으로 commit여부를 결정할 수 있지만,**

**POP의 경우는 판단할 수 없다.**

**(주소는 계속 재사용되므로, 단순히 Top을 넣었다 다시 넣은 경우에는 상관없다.)**

**만약 내부적으로 자료구조의 변경이 있었더라면, Top의 next가 변경된 경우이다.**

따라서 백업된 Top의 next에는 이전 주소값이 들어가있는 이상태에서,

**POP을 진행한다면 next는 엉뚱한 값을 가리키게되고 스택은 깨지게 된다.**

**PUSH의 경우 단순히 노드를 추가하는것으로, Top의 next가 무엇으로 바뀌든 상관없지만,**

**POP의 경우 Top을 제거해버리는 것이기 때문에 next값이 이전과 같아야 한다.**

(LockFree Stack의 경우에는 구현이 크게 어렵지않다. Queue가 문제…)

**ABA문제의 해결**

**1. 포인터를 유니크하게 사용하자**

포인터 메모리의 일부구간의 비트를 다른 용도로 사용하자.

(마치 SessionID로 StackIndex를 한방에 찾기위해 별도의 비트를 삽입한 것처럼)

이경우 32비트에서는 실패할 가능성이 높으며(우리는 32비틑 사용하지않긴 하지만),

**64비트라고 하더라도 100%안전한 것이 아니기 때문에 권장하지않는다.**

**2. Top과 Top->pNext 를 더블 CAS로 가자?**

변수 두개를 원자성이 보장되게끔 동시에 InterlockedCompareExchange할 수 있는건 없다.

대신 큰 변수를 대상으로 CAS를 진행하는 것은 가능하다.

**Top은 클래스의 멤버로 heap에 존재하고,**

**Top->Next는 노드의 멤버로 전혀 다른 위치에 존재한다.**

**이렇게 동떨어진 메모리들을 CAS시켜주는 것은 불가능하다.**

우리에겐 InterlockedCompareExchange128버전을 이용하여 64비트 변수 두개를 합쳐 CAS시키는것이 유일한 해결책이다.

**128bit크기의 변수를 대상으로 atomic한 연산이 보장되지않는다면 락프리는 구현이 불가능.**

(애초에 락프리는 이론적으로 구현이 불가능하다. 하드웨어적인 제약이 있기에 가능한 것.)

**2. next포인터를 얻어 보관하고, 이를 기준으로 더블 CAS.**

이것도 말은된다. 시도해보고 싶다면 시도해볼 것.

**단, 이경우 pop을 하고자할 때 TopNode->NextNode->NextNode를 저장해야하므로 구조가 복잡해질수 있음.**

**3. 별도의 Count를 두고 더블CAS진행 (사용할 방법)**

**Top과 Top->next를 동시에 CAS하는 것은 불가능하므로, Top->next의 변화유무가 아닌 자료구조 자체의 변화유무를 판단한다.**

따라서 Top의 주소를 저장하는 포인터를 두배크기인 128bit(16byte)로 만든다.

|  |  |
| --- | --- |
| 사용 주소 | 식별자(Stemp) |

**이후에 InterlockedCompareExchange128 으로 비교할 것인데, msdn에서 사용법을 정확하게 파악하고 사용할것.**

**(추가적으로, push의 경우 ABA문제가 일어나지 않으므로 더블 CAS가 필요없을 것)**

**노드생성 방식(new - delete)**

일단은 내부에서 new로 노드를 생성할 것.

Node\* nNode = new Node;

우리가 락프리를 만드는 이유는 락을 걸지 않기위함이다.

**그런데 new/delete는 thread-safe하도록 만들어졌기때문에, 내부에서 동기화가 일어난다.**

나중에 우리의 메모리풀도 락프리스택으로 바꿀 것이다.

**락프리가 완성되면 new/delete를 없애기 위해 락프리스택 내부에 노드메모리 관리자로 메모리풀을 넣을 것.**

지금은 아무것도 만들어진게 없으므로 템플릿-락프리스택으로 만들고 내부에서 new/delete한다.

new는 최초 한번만 하면 되므로, '루프돌릴때마다 new, 실패시 delete'의 행위는 하지않는다.

(추가적으로 new/delete를 사용할 때 문제가 나는 상황이 존재한다. 이는 직접 겪어보길 바라는 바이며, 추후에 언급할 것.)

락프리가 완성되었다면 테스트를 돌려본다.

**락프리 자체에서 문제가 나온다면 그냥 프로젝트를 중단하는게 나을정도로 난이도가 엄청나기 때문에 확실하게 테스트할 것.**

**더블 CAS**

윈도우에서 더블CAS는 InterlockedCompareExchange128().

당연히 기본 InterlockedCompareExchange()보다 느릴 것.

모든 CPU는 설계상 기본비트의 2배까지 커버하는 레지스터를 가지고있다.

우리는 64비트 기준으로 가기 때문에 128비트의 atomic한 연산을 지원한다.

**만약 이와 같은 하드웨어 상황이 받쳐주지않는다면 락프리는 구현이 불가능하다.**

**경계에 서는 개념**

**인터락함수의 전제조건. 경계에 서야한다.**

**일반적인 Interlocked함수들은 변수의 경계가 맞지않는 경우 atomic 연산을 보장해주지않지만,**

**Interlocked128은 내부에서 Crash가 나게된다.**

우리는지금까지 Interlocked로 도배했지만, 명시적으로 변수의 경계를 조절한 적은 없다.

컴파일러는 시작점이 0이 라는 전제로 가고있으므로, 인위적으로 패딩을 건드리지않는 이상은 변수마다 본인 자리에 맞게 경계를 잡아주기때문.

(만약 애초부터 시작점이 틀어져 4byte경계에 서있었다면, 이는 보장받지 못할 것.)

**경계를 명시적으로 잡기위해서는, 표준으로 정의된 alignas(n) / \_aligned\_malloc(n)을 사용한다.**

**Stack/malloc 할당시 시작점 경계**

그렇다면 Stack메모리와 malloc하여 뱉어주는 힙메모리는 몇을 기준으로 뱉어주는가?

Stack/malloc은 데이터 할당시 시작지점을 0으로 잡고 시작한다.

**만약 기준점이 4byte인 경우, 뒤에 할당되는 변수들의 경계선이 밀릴 수있다.**

**구조체/객체 할당시 시작점 경계**

**구조체 선언시 alignas(n)키워드를 변수에 선언할 수있고, 구조체 자체에 선언할 수있다.**

보통은 구조체의 선언부를 기준으로 잡는다.

16바이트 경계에 서고싶다면 구조체 선언자체를 16byte에 서게끔 해야한다.

**함수 호출시 매개변수의 시작점 경계**

아마도 될 듯. 확인해볼 것.

**결론**

**따라서 이러한 상황을 만들지 않기 위해서는 시스템비트의 두배로 경계를 잡고,**

**구조체에 8바이트변수 두개를선언/ 인자2개인 배열로 16바이트 할당하여 가야할 것.**

우리가 ABA문제로 Interlocked128을 사용하기위해 16바이트 경계에 변수를 세우려고 하고있다.

이때, push같은 경우는 ABA문제가 나지않지만 Top자체가 바뀌기 때문에 어쩔수없이 16바이트에 서게된다.

**Lock-Free 에러상황**

락프리 스택에 push/pop을 반복하면서 스택의 크기를 확인하고, 데이터역시 검증해야한다.

**1. 메모리 참조 오류**

**보통 락프리스택의 문제는 보통 ABA문제의 메모리참조 오류(잘못된 메모리접근, 잘못된 delete를 시도)가 날 가능성이 제일 크다.**

**2. 내가 A노드를 pop했는데, 다른쪽에서 또 pop해가는 문제**

**3. 데이터 유실 (Push했는데 데이터 사라진 상황)**

1000개를 넣었는데 1개가 없어지는 것

1번과 같은 메모리참조 오류는 VS에서 자체적으로 Crash를 내기 때문에 테스트하기 어렵지않지만, 2,3번의 경우는 우리가 직접확인해야 한다.

**Lock-Free 테스트 프로그램 구상**

**위와 같은 문제를 해결하기위해서는, 결국은 요소 하나하나를 확인하는 수밖에 없다.**

(1000개를 넣으면 1000개를 비교해야 한다.)

요소값이 재사용되는지에 대한 여부를 완벽하게 찾기는 힘들다.

최대한 완벽에 가깝게 테스트 하는 것

LanServer에서도 경험했지만, 락프리에서는 더더욱 브레이크로 하는 디버깅은 불가능해진다.

**더군다나 I/O작업이 일어나면 문제가 나지않기 때문에 로그출력도 사용하지못한다.**

**(2) 같은 데이터를 대상으로 두번 pop되는 문제**

**배열을 이용?**

배열Index의 경우는 빈값을 찾아내기 너무 힘들다.

아무도 사용하지않는 요소를 완벽하게 동기화잡아서 찾는 것이 굉장히 까다롭다.

InterlockedIncrement()역시 증감하는 그 순간에 누군가 치고들어올 수 있으므로 해결되지않는다.

**리스트를 두고 동기화사용?**

바깥에서 리스트를 하나두고 확인하기위해서는, 동기화가 필요하기 때문에 너무 느려서 테스트가 되지않는다.

결국 pop한 데이터에 대해 누군가 사용했는지 변화유무를 감지하는 방식으로 간다.

**따라서 데이터를 Interlocked으로 한쪽에서만 접근하여 변경할 수 있는 상황으로 만든다.**

**따라서 alloc을 받고 +1한다음, 잠깐의 대기시간을 거쳐 비교한다음 다시 원래값으로 돌려놓는다.**

**(3) 데이터 유실 (Push했는데 데이터 사라진 상황)**

스레드를 모두 끄고 노드개수를 확인하는 것은 테스트회수가 너무 적기 때문에 좋지않은 방법이다.

우리의 메모리풀은 FreeList이므로, 중간중간 AllocCount/UseCount를 체크한다.

push데이터가 유실됐을경우 최대치보다 적은수가 나올 것.

또는 pop이 두번된경우 최대치보다 많은 수가 나올 것.

또는 메모리가 없어서 내부에서 Crash가 날 것.

위 상황들을 지켜보며 테스트한다.

**Lock-Free 테스트 프로그램구현**

먼저, 스레드하나당 n개의 크기를가진 Stack을 가진다.

**스레드마다 서로 n개를 pop -> 그에대한 검증 -> 다시 push.**

위 과정을 루프로 해결한다.

**총데이터는 10만개 이므로 락프리스택의 크기가 n\*10 개를 넘어섰다면 이는 잘못된 것.**

**또한 모든 스레드가 동시에 pop한다고 하더라도 락프리스택이 empty인 상태에서 pop을 하는 경우또한 잘못된 것.**

먼저 구조체를 하나 만든다.

**struct st\_DATA**

**{**

**LONG64 Data;**

**LONG64 Count;**

**};**

// 여러개의 스레드에서 st\_DATA 데이터를 일정 수치만큼 생성(확보)

**테스트를 시작하기전에 해당 구조체를 동적할당 해서 준비한다.**

**숫자가 무조건 많다고 해서 테스트가 잘되는 것은 아니므로, 스택에 들어갈 수치는 계속 바꿔가면서 테스트 해야한다.**

0x0000000055555555 값의 고정된 변수로 시작하여, 스택에 push한다.

(스택에 들어있는 것은 모두 이 초기값을 가지고 있을 것)

약간 대기했다가 내가 넣은만큼 pop한다.

이때, 내가 넣은 변수를 다른스레드 쪽에서 pop해갈 수도있다.

---------------------------------------------------------------------------------

// 여러개의 스레드에서 st\_DATA 데이터를 일정 수치만큼 생성(확보)

**struct st\_DATA**

**{**

**LONG64 Data;**

**LONG64 Count;**

**};**

// 테스트 목적

// - 넣은 데이터 개수와 뽑은 데이터 개수의 일치 확인

// - 데이터를 넣었다가 뽑은 뒤에 이 데이터를 다른이가 메모리를 사용하는지 확인 (2중으로 뽑히는지 확인)

// 모든 데이터는 0x0000000055555555 으로 미리 초기화 함.

**// 0. 데이터 생성(확보)**

**// 1. Data = 0x0000000055555555 셋팅**

**// 1. lCount = 0 셋팅**

**// 2. 스레드 스택에 push(Alloc)**

**// 3. 약간대기**

**// 4. 내가 넣은 데이터 수 만큼 pop**

**// 4-1. - 이때 뽑히는건 내가 넣은 데이터일 수도, 다른 스레드가 넣은 데이터일 수도 있음**

**// 5. 뽑은 데이터가 초기값과 맞는지 확인. (큐(?)에 있는 데이터를 누가 사용하는지 확인)**

**// 6. 뽑은 데이터에 대해 값 변경.**

**// 6-1. 뽑은 데이터에 대해 lCount Interlock + 1**

**// 6-2. 뽑은 데이터에 대해 iData Interlock + 1**

**이는 일부러 사용을 해보는 것으로, 뽑고나서 사용을 하는데 다른쪽에서 그 포인터를 똑같이 뽑아갔다면 초기값에서 달라질 것이기 때문에 일부러 Interlocked로 변경해본다.**

**(이는 내가 pop했는데, 다른쪽에서도 pop을 해간상황을 가려내기 위한 항목이다.)**

그리고나서 일부러 사용하고 약간대기.

**// 7. 약간대기**

**대기시간은 yeildProcesser, Sleep(0), Sleep(1) 등으로 다양하게 조절해본다.**

**Sleep(1)도 사실 굉장히 긴편에 속함.(비권장)**

**// 8. + 1 한 데이터가 유효한지 확인 (뽑은 데이터를 누가 사용하는지 확인)**

그리고 1씩 증가시켰기 때문에 약간 대기한다음에 +1이란 데이터가 유효한지 볼겁니다.

다른쪽에도 이걸뭔가 차감을 했다거나, 다른쪽에서 증가를 더시켰다 한다면은 값이 달라질수있을거기 때문에.

**// 9. 데이터 초기화 (0x0000000055555555, 0)**

다시 Decrement해서 초기화한다.

**// 10. 뽑은 수 만큼 스택에 다시 넣음**

**// 11. 들어간 수 만큼 큐에서 다시 뽑아서 버림.**

**// 반복.**

**데이터를 초기화 -> 락프리에 push -> pop데이터 확인 -> 데이터변경(사용) -> 변경한 값이 유지되는지 확인 -> 데이터 초기화 -> 락프리에 push…**

새로 만드는 노드는 new를하든.. 아님 메모리풀을 사용하든 알아서.

더 좋은 방법이 있으면 선택해도된다.

ABA문제가 나기위해서는 pop을 두번이상하고, 전에 썼던 노드(메모리)가 push되어야 한다.

**지금 테스트방식은 n개를 모두뽑고, 넣어놓는 형태이므로 잘 안날 수있음.**

위에서는 스레드당 만개씩 10만개를 언급했지만, 개수에 따라 잘 나오는 문제들이 존재하므로

1,2,3개 등 다양한 테스트를 거쳐야 할 것.

**Delete를 넣는 경우**

**메모리 확보 -> 초기화세팅 -> 데이터 PUSH -> POP이후 사용 -> 잠깐 대기 ->**

**초기화하여 다시 PUSH -> POP이후 사용-> 잠깐대기 -> delete.**

실제 메모리를 delete를 할거냐말거냐 는 선택사항이다.

Delete를 하는 경우 실제 메모리 참조오류, 댕글링포인터 삭제오류 등이 유도될수 있음.

**테스트 환경 - 스레드 개수**

논리스레드 개수(하이퍼스레딩 포함)에 딱 맞추는게 가장 테스트가 잘될 것.

다수의 스레드를 사용하는 경우 실행되지못하고 ReadQ에서 기다릴 것이기 때문에, 스레드를 무작정 많이 쓴다고하여 테스트가 잘되는 것은 아님.

코어가 높을수록 문제가 잘나올 것.

쿼드코어 정도만 되어도 문제는 잘 나오는 편이다.

**push이후에 Sleep을 할수도 있는데, 이는 다른스레드에서 pop해가는 것을 유도하는것이다.**

**오류상황에서는 일부러 Crash를 내서 덤프뺴서 디버깅 할 것.**

ABA문제는 알아서 해결 해볼 것.

**오류가 난 상황은 모두 파악하여 왜 이런 오류가 났는지, 해결방법은 무엇인지 모두 숙지할 것.**

(ex 왜 next가 자기자신을 가리키고 있는지, 왜 next의 top이 깨졌는지..등)

락프리를 완성하면 스레드에 대한 감이 트일 것

락프리의 최종완성은 메모리풀이 완성되어야 한다.

**LockFree정리(메일)**

Lock-Free :  Lock 을 걸지 않음.  
  
일반적으로 Lock-free 구현에서 Spinlock 과 흡사한 방식을 사용.  
  
  
-. void Push(Node\* pNewNode)    
-. {    
-.  do {    
1.         Node\* t = pTopNode;    
2.         pNewNode->pNext = t;    
3.     } while( !CAS(&pTopNode, pNewNode, t) );    
4. }  
  
리스트 구조 스택에서 삽입을 간단하게 Lockfree 로 구현한 모습.  
  
위에서 CAS 는 Compare-And-Swap 의 약자로서 윈도우 API 에서는  
InterlockedCompareExchange 와 같음.  
  
1. 새로 위에 추가할 노드 pNewNode / t 는 최상위노드 (TopNode) 저장.  
2. 새로운 노드 (pNewNode) 의 다음에 t (TopNode) 를 연결  
3. TopNode 가 t 와 같다면, TopNode 를 NewNode 로 바꾼다.  
4. 3번이 실패했다면 다시 위로 1로 올라가서 재작업.  
  
1번에서 t 에 TopNode 를 저장한 뒤에  
새로운 노드의 Next 를 t 로 지정하여 리스트를 연결 하였음.  
  
마지막으로 TopNode 를 NewNode 로 바꾸어야 하는데  
다른 스레드에서 이미 TopNode 를 다른 노드로 바꿔버린 경우라면  
TopNode == t 가 성립이 안된다.  
  
이런 경우에는 다시 1로 올라가서 TopNode 와 NewNode 를 새롭게 연결.  
  
TopNode 가 여전히 t 의 상황이라면 TopNode 를 NewNode 에 대입하여  
최상위에 NewNode 를 넣기 완료!  
  
  
스핀락과 비슷하게 원하는 처리가 완료 될 순간이 올 때 까지 반복 작업을 함.  
  
  
스핀락은 단순하게 락 (대기) 를 위해 루프를 돌며  
락프리 알고리즘은 원하는 처리 자체를 위해 루프를 돌림.  
  
-----------------------------------  
  
  
# Lock free 구조에서 ABA 문제  
  
일부 lockfree 알고리즘에서 안전하게 비교를 했음에도  
다른 스레드에서 이를 사용.반환을 반복하므로서  
내부는 변경 되었으나 비교 대상은 일치하는 문제.  
  
Thread 1:                                 Thread 2:  
pop()  
read A from head  
store A.next `somewhere’  
  
                                          pop()  
                                          pops A, discards it  
                                          First element becomes B  
                                          memory manager recycles  
                                          ‘A’ into new variable  
                                          Pop(): pops B  
                                          Push(head, A)  
  
cas with A suceeds  
  
  
기본적인 락프리 구조에서 특정 하나의 변수를 비교하여  
해당 자료구조의 변경여부를 판단한다.  
  
  
  
  
  
Pop()  
st\_TOP\_NODE Top;  
do  
{  
    pPopNode = \_pTop;  
    pNewTop = \_pTop->pNext;  
} while ( InterlockedCompareExchange(&\_pTop, pNewTop, pPopNode );  
  
  
하지만 위의 코드는 스레드간에 동시에 사용시 아주 일부 문제의 가능성이 있다.  
이를 ABA Problem  ABA 문제 라고 함.  
  
Pop()  
do  
{  
    pPopNode = \_pTop;  
    <- 현재의 TOP 을 저장,  
    <- 마지막에도 TOP 의 변화가 없다면 이 스택 구조에  
    <- 변화가 없다고 확인하게 된다.  
  
    pNewTop = \_pTop->pNext;  
  
    <- 이 순간에 다른 스레드에서 두개 이상의 데이터를 뽑음.  
    <- Top 과 Top 의 다음이 빠져나갔음.  
    <- 그리고 다른 스레드가 먼저 뽑았던 (Top) 을 다시 Push 했음.  
  
    <- 결과 \_pTop 은 이 스레드 입장에서 변화가 없으나  
    <- \_pTop 의 Next 가 변화된 상황이므로 스텍이 엉망이 됨.  
  
} while ( InterlockedCompareExchange(&\_pTop, pNewNode, pNowTop );  
  
이를 해결하기 위한 방법으로는 더블CAS (2개의 데이터를 비교,교환)를 사용.

**<8.12>**

**락프리 자료구조를 구현하며 발생하는 여러가지 문제들**

**malloc()/free() 문제**

**1. malloc() / free()자체가 느림**

**2. 내부적으로 동기화가 걸리기 때문에 LockFree구현의 의미가 없어짐**

**3. CAS할 때 메모리참조 오류가 구조적으로 일어날수 밖에 없다.**

**(mallc/Free가 아닌 별도의 메모리프리리스트로 구현하는 이유)**

이 문제는 더블CAS보장(메모리가 연결되어있어야 함)이나, ABA문제랑은 무관하다.

**malloc()/free() 사용시 구조적인 문제**

락프리스택의 pop구현시,

**TopNODE\* tNode = this->TopNode;**

위와같이 TopNode를 백업시켜 로컬변수인 tNode를 기준으로 로직이 진행된다.

**그런데 tNode는 어디선가 pop하여 노드가 delete될 수 있기 때문에, 유효하지않다.**

**이 경우 tNode->NextNode를 접근하려고 하는 경우 메모리 참조 오류가 나게된다**

(Page-fault되지 않아 문제가 나지않을 수있지만, 언제 page-fault될지 알수없으므로 이에 의존적인 로직을 짜서는 안될것이며, delete를 사용하는 경우 0x08123으로 초기화되기 때문에 메모리참조오류가 100%일어나게 된다.)

**이 구조적인 문제는 락프리스택 내부에서 delete하기 때문에 생기는 것으로, 데이터 자체(<T>타입)을 전달해야 하는 락프리스택(또는 큐) 구조상 메모리 참조 오류를 피할 수 없다.**

**노드의 생성 / 삭제 문제**

**락프리스택을 구현하려면, 메모리참조오류가 필연적으로 발생한다.**

(A-B를가진 스택에서 A스레드가 A를 Top으로 저장하고 Free하려고했음. 그런데 어디선가 이미 해제해버린 노드인 경우 문제.)

**메모리풀은 free로 데이터가 들어온경우 숨겨둔 노드를 찾아 노드를 보관하므로, 삭제되지않음.  
따라서 누군가 이미 pop해버렸다고 하더라도 존재하는 노드이므로 참조오류가 나지않는다.**

그렇다면 스택을 메모리풀 방식으로 만들면 참조오류가 나지않는가?

**스택은 자료형자체를 주고받는다. 따라서 들어온 자료형을 보관하고싶어도 보관할수가없다.**   
(노드를 만들어 관리하고, 자료형만 빼고 넣는다면 결국 내부에 메모리풀을 두게되는것)

메모리풀은 포인터 형태이므로 앞뒤에 정보(노드)를 숨겨 보관하는것이 가능하지만, 스택은 불가능하다.

**따라서 별도의 메모리풀없이 malloc/free만을 사용해 락프리 자료구조를 구현하는 것은 구조적으로 불가능하다.**

**LockFree-MemoryPool**

락프리풀 Alloc = 자료구조 pop

락프리풀 free = 자료구조 push

**스택과 같은 구조지만 노드자체를 던지고 받음으로서 생성/삭제의 작업을 생략하여 독립적인 락프리구조의 구현이 가능해진다.**

**내부에 데이터가 없는데 pop하는 경우**

TopNode

{

NODE\* pNode = nullptr;

\_\_int64 Count = 0;  
}

우리는 최초 TopNode를 생성하여 head노드로서 가지고간다.

이때 pop할 데이터가 없는 상황에서 pop을 호출하는 경우,

**pNode의 pNext를 찌르는 쪽에서 메모리 참조 오류가 나게된다.**

로직상에서는 (TopNode->pNode = TopNode->pNode->pNext);

**실제 상황에서는 별도의 예외처리 코드로 막아야하지만,**

**우리가 진행하는 테스트 환경에서는 있을 수 없는 일이므로 일부러 메모리참조 오류가 나게끔하고 진행한다.**

**ABA문제 해결**

**1.별도의 Count를 두고 사용**

**ABA문제는 별도의 Count를 두고, push/pop할때마다 Count를 올려 변화가 있었음을 체크하는 방식을 주로 사용한다.**

**단, Count를 사용한다면 ABA문제가 아닌경우에도 변화가 감지되면 CAS에 성공하지못하고 빠지게된다.**

push같은 경우에는 Top의 next가 달라졌다고 하더라도 문제될게 없으므로 굳이 CAS128을 사용할 필요는 없다.

**2. Count대신 pNext포인터 값을 저장해두는 방법도 있겠다.**

Top노드에 pNext의 주소값을 따로 보관해두고, 나중에 CAS시 변화가 있었는지 비교하는 것.

pNext가 같다면 여러 번 push/pop이 일어났다고 해도 상관없는 상황이다.

단, pop에서 아래와 같은 상황이 일어난다.

TopNODE\* NewTop = alloc();

NewTop->pNode = pop->pNext;

NewTop->Unique = **pop->pNext->pNext;**

**위와 같은 경우 굉장히 지저분해지기 때문에, Count로 가는 것을 권장한다.**

**InterlockedCompareExchange128()**

BOOLEAN InterlockedCompareExchange128

(

LONG64 volatile \*Destination,

LONG64 ExchangeHigh,

LONG64 ExchangeLow,

LONG64 \*ComparandResult

);

LONG64 volatile \*Destination

대상값에 대한 포인터.

이 매개변수는 128비트 필드로 간주되는 두개의 64비트 정수배열.

LONG64 ExchangeHigh,

주소기준 높은 값

Dst[1]에 들어갈 값.

LONG64 ExchangeLow,

주소기준 낮은값

Dst[0]에 들어갈 값.

LONG64 \*ComparandResult

**성공여부와 관계없이 Dst의 원래값으로 덮어쓴다.**

**반환 값**

**값이 Dst != Comp 인경우 0을, 같은 경우에는 1을 반환한다.**

같은 경우는 Exchange변수가 Dst에 대입된다.

**비고**

**Dst값은 무조건 16바이트 경계에 서야한다.**

(우리가 선언할 수 있는 변수의 최대크기는 8바이트이므로, 8바이트 경계에 서게된다.)

그렇지 않은 경우 에러를 뱉어준다. 따라서 \_aligned\_malloc으로 할당해야 한다.

이 함수는 x64기반 시스템에서만 제대로 작동된다.

해당함수는 MemoryBarrier를 생성하여 메모리 작업이 순서대로 완료되도록 한다.

**메모리풀 락프리스택**

메모리풀 락프리스택 -> 락프리 스택(자료구조)

순으로 진행하므로, 먼저 메모리풀 락프리스택에 대해 완벽하게 테스트가 진행되어야 한다.

메모리풀 락프리스택을 테스트 하기전에, 먼저 main쪽에서

**해당 테스트프로그램에 할당개수만큼 동적할당하고, 초기값으로 셋팅한 다음 테스트를 시작한다.**

(초기값이 있어야 테스트가 가능)

메모리풀 락프리스택 역시 같은 방법으로 테스트가 진행된다.

**alloc(pop)하여 할당이 제대로 됐는지 확인.**

**n개를 alloc했을때, 멀티스레딩 환경에서 alloc이 n개를 유지하는지 확인.**

**데이터의 초기값은 0x0000000055555555로 할 필요는 없지만, 0000..값은 비권장.**

(캐시라인이 틀어지는 경우를 보고싶었지만.. 사실상 의미없음)

락프리 스택자체에 대한 문제를 보기위해서는 push/pop 내부에서 최대한 경합을 많이 발생시켜야하므로, 코어개수에 스레드를 맞춰 테스트하는 것을 권장한다.

**Sleep(1)을 넣는 경우는, 락프리스택 코드는 돌아가지만 포인터를 두개이상 뱉어주는 경우(같은 값이 두개이상 나온경우)**

**아주 약간의 시간을 넣어서 다른쪽 스레드에서도 alloc받은 해당 데이터를 사용하는 경우 값을 변경하여 식별하는 것임.**

**일단 스레드 CPU개수를 맞춰서 그대로 만들어서 Sleep()없이 돌린다.**

Alloc(pop)하여 데이터를 Increment했다가 다시 Decrement하여 비교하는 이유는, 다른쪽에서 같은 값이 뽑혀 값을 사용하는지를 확인하기 위한 절차이다.

모니터링 스레드에 해당 메모리풀에 대한 수치를 나타내야 할 것.

(사용중인 수치 / 내부에서 할당된 수치.)

**락프리에서 Empty()체크**

1. Top이 nullptr인지 아닌지 확인.

2. 내부에 할당된 StackSize로 파악.

메모리풀에서는 AllocCount - UseCount == 0으로 했지만.

락프리의 경우 문제날 것. 고민해봐야한다.

메모리풀은 여기서 한번더 개량해야 성능이 나온다.

**<8.17>**

**LockFree Queue**

락프리 스택의 경우 push에서는 ABA문제가 나지않는다.

락프리 큐역시도 위와 같은 상황이 존재한다.

하지만 왜 문제가 나지않는지 정확히 숙지하고 있어야 할 것.

락프리스택의 ABA문제는 pop에서 하나 존재하지만, 큐는 여러 개의 ABA문제를 가지고있다.

**LockFree - 더블리스트 형태**

**자료구조에 영향을 주는 작업일 경우 한번의 commit으로 완료되어야 한다.**

싱글리스트의 경우 사전작업이 자료구조에 영향을 주지않는다.

**하지만 더블리스트는 앞뒤의 노드변수를 동시에 바꿔줘야 한다.**

**이와 같은 이유로 LockFreeQueue역시 더블리스트로 구현이 불가능하다.**

**LockFree - 배열 형태**

배열은 저장공간이 정해져있고, front또는 pop이 Index를 가리키고 있는 형태이다.

**따라서 데이터를 삽입 또는 삭제할 때 한번의 행동으로 끝내지못하고, 두번으로 나눠져야한다.**

**1. 데이터를 push**

**2. Index의 증감/차감**

이 두가지를 동시에 commit할 방법이 없다.

**LockFree - Queue**

**Deqeuue상황**

락프리 큐의 Dequeue행위 자체는 락프리스택의 pop과 같다.

**단, 락프리스택은 TopNode->pNode == nullptr인 경우 empty()상태로 볼 수 있었다.**

**하지만 락프리 큐는 rear포인터가 따로 존재하여 무언가를 찌르고있는 상황이다.**

정리하자면,

**1. Front와 rear가 동시에 null을 가리키는 상황 (마지막 데이터 Dequeue시)**

**2. head/tail이 모두 null인 경우 head와 tail을 동시에 NewNode로 바꿔야 하는 상황 (첫 데이터 Enqeueu시)**

**따라서 front나 rear 둘중 하나만을 건드리는데도 다른쪽까지 변경이 되어야 하는 상황이다.**

이는 락프리 구조에서 허용되면 안될 것.

**따라서 락프리 스택의 pop방식으로는 락프리 큐의 Dequeue가 구현이 불가능하다.**

**그렇다면 더미를 사용한다면??**

**더미노드 유무의 차이점**

**1. head/tail이 포인터로 존재하는 경우**

맨처음에는 head와 tail이 nullptr을 가리키고 있는 상태로 있어야 한다.

첫 데이터를 Enqueue시 head와 tail이 NewNode를 가리키게 되고,

마지막 데이터를 Dequeue시 head와 tail이 nullptr을 가리켜야한다.

**2. head/tail이 실제 노드(더미)로 존재하는 경우**

하지만 head/tail이 실제 노드로 존재하는 경우라면 위와 같은 조건이 필요없어 진다.

head -> Node -> tail

Enqueue시 tail뒤에 노드를 삽입하면 될것이고,

Dequeue시 head->next를 삭제하고 head->next->next(tail)을 head->next를 연결해주면된다.

**따라서 더미를 사용한다면 nullptr인지 체크할 필요가 없어지기 때문에 로직이 간단해진다.**

**Dummy LockFree - Queue**

먼저 일반적인 방식의 더미를 넣어 시뮬레이션을 해보자.

push\_back, pop\_front를 가정하고 진행한다.

**Enqueue()**

위 로직에서 Enqueue(Node2)를 한다면,

**head -> Node1 -> tail + Node2추가.**

**1.Node2의 Next를 tail에 연결 (Node2->Next = tail)**

**2.Node1의 Next를 Node2에 연결 (Node1을 알수없음)**

**1번의 경우는 문제없지만, 2번의 경우 Node1을 알수있는 방법이 없다.**

더블리스트 큐형태를 한다면 구현이 가능해지겠지만 락프리는 더블리스트 적용이 불가능한 상황.

**Dequeue()**

**head -> Node1 -> Node2 -> tail**

위 로직에서 Dequeue를 진행한다면,

**1. head의 next를 Node2와 연결한다. (head->Next->Next = head->Next);**

**이때 head->Next->Next가 변할 수 있는 상황이나, DCAS로 진행하면 문제없음.**

따라서 Dequeue의 경우는 문제없음.

**일반적인 Dummy방식을 이용한 LockFree-Q 결론**

**Dequeue의 경우는 문제가 없지만 Enqueue의 경우 tail의 앞노드를 찾아낼 방법이 없으므로 구현이 불가능하다.**

**LockFreeQ에 더미를 적용하는 방법**

그래서 선택하는 방법은,

**head는 더미가 있지만 tail은 더미가 존재하지않고 마지막 데이터를 찌르는 형태로 간다.**

**Tail을 포인터형태로 더미를 가리키게 바꿨기 때문에, Enqeueu시 rear->next에 NewNode를 추가하고 tail를 이동시키는 방식으로 tail의 앞노드를 알수없는 문제를 해결했다.**

하지만, 큐내부에 데이터가 없는 경우 tail는 nullptr을 가리키게되고,

결국 head->Next는 nullptr이 되므로

**head를 건드렸는데 tail까지 건드려야하는 처음 문제로 되돌아간 상황이다.**

head\* tail\*

↓ ↓

Dummy -> NodeA -> nullptr

**Enqueue 문제없음**

**Dequeue**

head -> next = head->next->next;

tail = tail->Next

**결국 head나 tail하나만을 사용하는데도 다른쪽이 변경이 가해지는 상황으로, 이전에 더미를 사용하지않아서 생기는 문제와 동일하다.**

따라서 이 상태로만 봐서는 처음에 났던 문제가 다시 생기게된다.

((

**앞서 더미가 없는 LockFreeQ의 경우의 문제 상황**

**1. Front와 rear가 동시에 null을 가리키는 상황 (마지막 데이터 Dequeue시)**

**2. head/tail이 모두 null인 경우 head와 tail을 동시에 NewNode로 바꿔야 하는 상황 (첫 데이터 Enqeueu시)**

**-> 위 두경우는 front나 rear 하나만을 사용하는데도 다른쪽에 변경이 가해지므로 락프리구조에서 사용이 불가능했다.**

))

**중간정리 (락프리 큐 구현 시 문제)**

(head로 디큐, tail로 인큐)

**1. 더미 없이 구현**

head와 tail이 동시에 null을 가리킨다.

첫 Enqueue시 둘다 변경, 데이터가 1개일때 Dequeue시 둘다 변경.

**-> head, tail중 하나만 건드려도 두개를 모두 변경해야 하는상황.**

**2. 더미사용해서 구현**

**(1)문제가 해결됨. 추가로 nullptr체크가 필요없어졌으므로 로직이 간편해짐.**

Dequeue는 문제없으나 Enqueue에서 문제가생김.

**Enqueue**

----------------------------------------------------

**A -> N1(새노드) -> Tail(더미)**

**N1->Tail을 문제없음.**

**A->N1을 해야하는데, A를 알수가없음**

----------------------------------------------------

**3. head만 더미를 사용해서 구현**

(2)문제가 해결되지만 (1)문제가 다시발생

인큐문제없음. 디큐할때 둘다 한쪽만 사용해도 둘다변경해야함

head\* tail\*

↓ ↓

Dummy -> nullptr

Dequeue

headDummy->Next = headDummy->Next->Next;

tail->nullptr;

4. 마지막에 썼던 노드를 더미로활용

1,2문제를 모두해결함.

**더미를 적용한 최종적인 LockFree-Q**

따라서, 다음과 같은 조건들로 해결을 한다.

**1. Front는 포인터 형태로 존재한다. (이전에는 head자체가 더미)**

**2. Front는 Dummy를 가리키고, 생성자에서 Dummy를 생성한다.**

**3. Dummy데이터 뒤에 실제 데이터노드가 이어진다.**

**4. tail역시 포인터 형태이며, 마지막 데이터노드를 가리킨다.**

**5. 더미는 고정으로 두지않고, 마지막에 썼던 노드를 다시 더미로 활용한다.**

**Front\* rear\***

**↓ ↓**

**Node1(Dummy) -> Node2 -> Node3 -> Node4**

**위 상황에서 Dequeue를 하게되면 front가 next를 타고가고, Node1이 더미노드가 된다.**

**Node1의 데이터는 이미 빠져나갔지만, 자료구조에서는 여전히 더미로 존재하고 있는상황.**

**Front\* rear\***

**↓ ↓**

**Node2(Dummy) -> Node3 -> Node4(rear);**

**위와 같은 형태를 취함으로서 Dequeue를 했음에도 하나의 더미를 유지하면서 nullptr을 가리키지 않게 되었다.**

**따라서 더미노드가 항상 하나는 존재해야하므로, 처음 큐를 만들 때 더미를 하나 만들어둔다.**

front\* rear\*

↘ ↙

Dummy

Enqueue()할 경우 새로운노드를 추가하고, (Node->pNextNode = Node1)

Rear를 옮긴다. (rear = Node1)

front\* rear\*

↓ ↓

Dummy -> Node1

이시점에서 Dequeue()하는 경우 Dummy의 next를 뽑아서 내보낸다.(Dequeue)

그리고 front를 Dummy의 next, Node1로 대입하고, 기존에 있던 Dummy(Node)를 지운다.

front\* rear\*

↘ ↙

Node1(Dummy)

**따라서 front는 항상 Dummy를 가리켜야하며, 실제 노드의 삭제는 것은 한박자 느리게 진행된다.**

**위와 같은 형태를 취함으로서 front와 rear가 절대 nullptr을 가리키지 않게끔 되었다.**

front\* rear\*

↓ ↓

Dummy -> Node1

**락프리 스택의경우, 노드를 생성하여 연결하고 Top을 교체하는것으로 commit되었다.**

**하지만 락프리 큐의 경우는 해당 방식이 Dequeue는 가능하지만, Enqueue가 불가능하다.**

**Enqueue/Dequeue의 간단한 구현**

Dequeue

if(pheadNode->EndNode != pTailNode->EndNode) { 디큐가능 }

Enqueue

1. if(tail->pNext == nullptr) tail ->next = NewNode

2. if(tail ->pNext == NewNode) tail = NewNode;

**Enqueue 의 과정**

Stack의 경우 NewNode의 next가 Top을 가리킨다고 하더라도 자료구조에 아무런 영향이 없었다.

**하지만 큐의 경우는 두개의 작업 모두가 자료구조에 변경을 가하는 형태이다.**

**(1) tail->pNext = NewNode;**

**(2) tail = tail ->pNext;**

어떤 행위를 먼저하든 자료구조에 변경이 되는상황이고, 이 두행위를 DCAS하는 것은 불가능하다.

head\* tail\*

↘ ↙

Node(Dummy)

그래서 인큐의단계를 두단계로 나눈다.

**1. 새노드를 tail->pNext를 연결시키는 행동**

**2. tail 를 새노드를 가리키도록 변경하는 행동 (tail = tail ->pNext;)**

**1. 새노드를 tail->pNext를 연결시키는 행동**

**일단 tail->pNext에 연결시키는 행위가 성공한다면 Enqueue로 가정한다.**

**Dequeue라는 행위는 front를 타고 next로 내려오는 것이기 때문에 문제없음.**

**결국 CAS를 통해 tail->pNext가 NULL인경우 NewNode를 Enqueue할 것인데,**

**이때 tail이 삭제되어 nullptr이라면 메모리 참조 오류가 생기게 될것이다.**

(대상노드(ex TopNode)그 자체 포인터의 변화가 있냐없냐를 확인하는 것이면 문제가 없음. 이는 tail자체에 접근이 아닌 tail->pNext에 접근하면서 나는 문제)

---------------------------------------------------------------------------------

**Stack - pop에서도 존재하는 메모리 참조 오류**

New/delete를 사용하는 Stack의 pop()내부에서도 이와 같은 비슷한 문제가 존재했다.

pop을 하는경우 TopNode = TopNode->pNext 이므로,

CAS의 Exchange인자로 TopNode->pNext가 들어가게된다.

**이때 로컬로 복사해둔 TopNode이 다른쪽에서 이미 pop되어 delete되었다면 TopNode는 null이되므로, 메모리참조 오류가 나게된다.**

스택에서는 이러한 문제를 해결하기위해 메모리풀을 사용하여, delete가 되지않게끔 했다.

---------------------------------------------------------------------------------

**스택의 pop에서는 락프리 메모리풀을 활용하여 이러한 문제를 해결했다.**

**큐 역시도 메모리풀을 활용한다면 간단하게 해결된다.**

**메모리풀 을 사용한다면 모든 노드가 내 관리하에 있기 때문에,**

**단순히 null인지만을 확인하고 CAS를 진행한다면 아무 문제가 없을 것이다.**

(애초에 null로 뭔가 판단해서 들어가는 경우 메모리풀이 전제되어야한다.)

락프리스택에서는 내가 찌르고 있던 포인터가 해제되는 것이 문제였다.

**락프리 큐의 경우는 해제도 역시 문제고, 해제가 되지않고 존재하는 메모리라고 하더라도 비교 조건 자체가 if(rear->pNext == nullptr)이기 때문에 엉뚱한 메모리에 추가될 가능성이 있다.**

**따라서 메모리풀을 반드시 사용해야 구현이 가능할 것.**

**코드상 구현**

**---------------------------------------------------------------------------------**

EndNode CompNode; //CAS에서 비교할 노드

NODE\* pNextTail; //NextTail을 보관할 노드

**//비교할 노드를 Local로 선언하여, 현재 tail노드를 백업한다.**

**CompNode.UniqueNum = \_pTailNode->UniqueNum;**

**CompNode.pNode = \_pTailNode->pNode;**

**// CAS 직전에, NextTail을 보관할 별도의 노드에 CompNode(nullptr)을 보관한다.**

**// 이후 nullptr을 체크한다음 Enqueue를 진행한다.**

**pNextTail = CompNode.pNode->pNextNode;**

**if(pNextTail == nullptr)**

**CAS(…); (tail->pNext를 바꾸는 것)**

...

**---------------------------------------------------------------------------------**

CompNode는 로컬변수로 두고, 해당 노드 내부의 변수들을 \_pTailNode변수로 채워넣었기 때문에 이 값들은 변하지 않을 것이다.

**따라서 CompNode.pNode->pNextNode와 이를 대입받은 pNextTail은 nullptr일 것이다.**

**불필요해보이지만 위 로직으로 인해 DCAS의 호출량을 줄일 수 있음.**

**(만약 다른쪽에서 tail을 바꿔 pNextTail이 null이 아닌경우에는 어차피 CAS에서 실패될 것)**

**2. rear가 새노드를 가리키도록 변경하는 행동**

**앞서 tail의 next가 null인경우 nNode를 꽂아줌으로서 Enqueue의 절반이 끝났다.**

그다음 할일은 rear가 새노드를 가리키도록 변경하는 것.

따라서 CAS가 두번이 들어가는 꼴이된다.

**이 경우 당연히 첫 CAS가 실패하는 경우 Enqueue는 실패하여 다시 돌아간다.**

**하지만 두번째 CAS는 실패 여부를 판단하지않고 그대로 빠져나온다.**

(코드상에서 보자면, 내가 rear를 미는행동을 실패했다고 하더라도 그대로 빠져나오는 것.)

(실패했다고해서 돌이킬수있는 방법도 없음)

**따라서 첫 CAS가 성공한다면 Enqueue는 성공으로 간주한다.**

만약 2번째 CAS가 실패한다는 것은

-----------------------------------------------------------------------

front\* rear\*

↓ ↓

Dummy -> Node1 (+ NewNode) 추가할 예정

rear->next = NewNode (1) (새로운 노드 삽입)

rear = rear->pNext(NewNode) (2) (rear이동)

결국 (2)는, CAS(rear, NewNode, Node1)와 같은 형태로 rear를 바꾸게 될것이다.

↑ ↑ ↑

(Dst) (Exch) (Comp)

-------------------------------------------------------------------------------

(2)와 같은 CAS를 실패했다는 것은 Dst와 Comp가 같지 않다는 얘기이므로, 이 사이에 누군가 노드를 추가했다고 볼 수있다. (rear가 바뀐상황)

**지금 이 상황으로 만봐서는 도중에 rear가 바뀌는 상황은 없을것같지만, 이런 상황이 나오게되므로 충분히 고민하고 왜 이런상황이 나오는지 정확히 숙지하고 있어야 할 것.**

**2-1. 새노드 추가후 tail을 변경하는 것이 실패하는 경우**

이 문제는 ABA문제가 아니며 위와 같은 환경에서 나올수있는 특수한 상황이다.

**연속으로 Enqueue하여 tail을 옮기기전에 뒤쪽에 노드가 두개 붙었다?**

----------------------------------------------------------------------------------

head\* tail\*

↓ ↓

Node1 -> Node2 -> Node3(NewNode) 추가됨

(Dummy)

----------------------------------------------------------------------------------

먼저 Node3을 뒤에붙이는 DCAS를 성공했다고 가정한다.

이제 tail을 Node3로 바꾸려고 하는데, 다른곳에서 Enqueue가 진행된다면?

(또는 Node3을 DCAS하기직전까지 로직이 다른스레드에서 진행됐다면?)

**CAS - if(tail->pNextNode == backup\_pNextNode) tail->pNextNode = 새노드;**

----------------------------------------------------------------------------------

head\* tail\*

↓ ↓

Node1 -> Node2 -> Node3 + Node4 (추가하려고함)

(Dummy)

----------------------------------------------------------------------------------

애초에 UniqueCount가 달라졌기 때문에 Node4를 붙이는 것은 실패한다.

만약 DCAS가 아닌 CAS를 사용해서 UniqueCount를 확인하지않더라도,

**Tail->next == null일 경우에만 Enqueue가 성공함.**

**정리**

**첫번째 CAS(노드추가 성공) 이후에는 어떤 스레드에서든 Enqueue에 성공하지못한다.**

**Tail->next == null일 경우에만 Enqueue가 성공하기 때문이다.**

그렇다면 어떤경우에 2번째 CAS(tail을 뒤로밀기)가 실패할까?

시뮬레이션 가정

**스레드A 스레드B** **진행된 일**

**스레드 A**

----------------------------------------------------------------------------------

head\* tail\*

↓ ↓

Node4 -> Node5 + Node6(NewNode) 추가할 예정

(Dummy)

**--------(Node5백업)**

----------------------------------------------------------------------------------

이상황에서 tail(Node5)의 next가 null인지 판단한다음 Enqueue를 진행 할 것이다.

따라서 tail인 Node5를 백업한다.

**스레드 B**

**----------------------------------------------------------------------------------**

**head \* tail \***

**↓ ↓**

**Node4 -> Node5 -> Node7 -> Node8**

**(Dummy)**

**----------------------------------------------------------------------------------**

2번의 인큐가 성공했고, tail을 미는것까지 성공 (Enqueue완전히 성공)

**스레드B**

**----------------------------------------------------------------------------------**

**head \* tail \***

**↓ ↓**

**Node4 -> Node5 -> Node7 -> Node8**

**(Dummy)**

**(Node4, Node5 완전히 삭제)**

**----------------------------------------------------------------------------------**

두번의 디큐를 거쳐 Node7이 Dummy가 된다.

**스레드B**

**----------------------------------------------------------------------------------**

**head \* tail \***

**↓ ↓**

**Node7 -> Node8 -> Node5**

**(Dummy)**

**----------------------------------------------------------------------------------**

새로이 Enqueue했는데, Node5가 재사용된다.

Tail은 아직 밀지못한 상태.

**스레드 A**

**----------------------------------------------------------------------------------**

**head \* tail \***

**↓ ↓**

**Node7 -> Node8 -> Node5 -> Node6**

**(Dummy)**

**(Node5백업 : Node6 추가할 예정)**

**----------------------------------------------------------------------------------**

A스레드에서 Node5->pNext == nullptr 이므로 뒤에 노드를 추가한다.

따라서 Node6이 새로이 추가되었고, **tail 미는것을 실패한다.**

이 문제가 자료구조로서 결함인지 아닌지 결론내고 결함이 있다고 생각한다면 고쳐야 할 것.

**이와 같이 tail을 미는 것을 실패하는 경우가 생김.**

이는 ABA문제가아닌 락프리 큐만의 독특한 상황.

ABA문제는 UnqiueCount를 사용함으로서 해결가능.

**락프리 Q의 ABA문제**

**스레드A**

**----------------------------------------------------------------------------------**

**head \* tail \***

**↓ ↓**

**Node1 -> Node2 -> Node3**

**(Dummy)**

**----------------------------------------------------------------------------------**

Dequeue를 하기위해 들어옴

1. head(Node1)백업. **(UniqueCount백업을 하지않고, 어떻게ABA문제가 나는지 파악.)**

-------------------------------------------여기서 멈춤

**2. if(head == 백업노드) head->pNext = head;**

Node1삭제, Node2의 데이터를 Dequeue, Node2를 더미로 삼을 예정

**스레드B**

**----------------------------------------------------------------------------------**

**head \* tail \***

**↓ ↓**

**Node1 -> Node54 -> Node55**

**(Dummy)**

**----------------------------------------------------------------------------------**

Dequeue/Enqueue가 반복되다 Node1이 다시 head로 왔다고 가정

**스레드B**

**----------------------------------------------------------------------------------**

**head \* tail \***

**↓ ↓**

**Node1 -> Node54 -> Node55**

**(Dummy)**

**----------------------------------------------------------------------------------**

Dequeue를 하기위해 들어옴

1. head(Node1)백업. **(UniqueCount백업을 하지않고, 어떻게ABA문제가 나는지 파악.)**

-------------------------------------------여기서 멈췄다가 다시시작

**2. if(head == 백업노드(Node1)) head->pNext = head;**

Node1삭제, Node2의 데이터를 Dequeue, Node2를 더미로 삼을 예정 -> 자료구조 깨짐

2번 조건에 부합하므로, CAS성공.(자료구조 깨짐)

**LockFree 테스트 방향**

락프리 큐 같은 경우 많이 넣고 빼는 것이 의미없음.

주로 Head와 Tail이 만났거나 안만났을 때 문제이다.

(스택은 Top이라는거 하나만 건드리기 때문에 많이 넣던 한 개넣던 증상이 다 똑같음)

결국 누군가 해제한 메모리를 재사용하면서 나타는 문제임.

따라서 큐에대한 테스트는 볼륨을 굉장히 작게가져간다.

스택/메모리풀처럼 10000개씩 가는 것은 head와 tail이 벌어져있기 때문에 테스트가 잘 안될 것.

사실 사용하는 쪽에서(외부) Increment/Decrement해서 답을 비교하는 곳에서는 문제가 날 가능성이 거의없음.

그전에 head나 tail이 이상해지는 등의 상황이 생긴다.

**InterlockedCompareExchangePointer**

PVOID InterlockedCompareExchangePointer(

PVOID volatile \*Destination,

PVOID Exchange,

PVOID Comperand

);Interlocked

X64환경에서는 64bit크기로, x32환경에서는 32bit크기로 원자적인 수행을 하는역할.

서버는 무조건 64비트 환경으로 가기 때문에 interlocked64와 interlockedpointer어느것을 써도 상관이 없다.

**LockFreeQ - Dequeue()**

Dequeue는 Enqueue에 비해 큰 특이사항이 없다. 한번 구현해볼 것.

Dequeue하기전에 뽑아낼 데이터가 있는지 먼저 체크한다.

**LockFreeQ - Empty()**

데이터가 얼마나 있는지 체크하기위해서 두가지 방법을 사용할수 있을 것이다.

**1. 멤버변수의 락프리큐 size를 보고 판단하는 방법**

**2. (head.pNode == nullptr)를 체크하는 방법**

**일단은 size를 체크하여, headNext가 nullptr인지 판단하고 size가 0인지 확인하여 비었는지 확인하는 방법으로 진행하여, 어떤 문제가 일어나는지 파악해 볼 것.**

(위와 같은 문제의 원인과 해결방안을 모두 숙지하고 있어야 한다. Size를 사용하는경우, Size가 1이라도 있는경우 100% 성공해야 한다.)

Dequeue 시 Empty()의 여부를 판단할 때,

1. size의 크기로 판단한다면 pop시 차감하고 들어온다음 내 시도는 반드시 성공해야 할것.

2. TopNode.pNode == nullptr로 판단한다면 pop이 성공한다음 Decrement해야 할 것.

**LockFree Q의 사용**

LockFree큐가 완성이 된다면 LanServer에 Send.RingBuffer를 락프리 형태로 바꿀 것이다.

Send용 링버퍼가 직렬화버퍼 포인터를 다루는 템플릿 락프리 큐가 되는 것.

Recv용은 링버퍼 그대로 진행한다.

그리고 SessionArrayIndex를 관리하는 스택을 락프리 스택으로 바꿀 것이다.

언급했듯, 자료구조로서 큐보다는 스택이 훨씬 가벼우므로 입출력에 관한 순서가 중요하지않다면 스택을 선택하는게 좋다.

LanServer는 따로 속도 테스트는 하지않는다.

또한 단순히 CPU사용률만 가지고서 뭔가를 판단하는 것은 애매한 부분.

CPU사용률을 많이 먹으면서 빨리 처리하면 그게 더 좋을수도 있기 때문.

**Lock-Free Stack에서의 문제**

스택내부에서 Comp노드를 셋팅할때, 8바이트씩 변수를 나눠 대입한다.

(현재 16byte를 한번에 읽을 방법이 없기 때문)

**(ㄱ) TopNode.pNode = this->\_TopNode->pNode;**

**(ㄴ) cTopNode.UniqueCount = this->\_TopNode->UniqueCount;**

**이때만약 (ㄱ)->(ㄴ)순서로 간다면 ABA문제가 발생했음에도 감지하지 못하는 문제가 생긴다.**

먼저 스레드1 에서, push를 호출하여 (ㄱ)구문 까지 진행되었다고 가정하자.

그리고 스레드2 에서 pop을 호출하여 UniqueCount를 증가시켰다고 가정하자.

**다시 스레드1에서 (ㄴ)을 읽게되면, 증가시킨 UniqueCount값을 읽어오기 때문에 Comp노드와 현재 노드의 Count값이 일치하여 ABA문제가 발생했는지 구별할 수 없게된다.**

**이는 락프리스택, 큐, 메모리풀 모두 생길수 있는 문제이므로 반드시 염두해 둘것.**

**따라서 (ㄴ)->(ㄱ) 서순으로 진행한다면 이와같은 상황을 막을 수 있다.**

**구조체 대입으로 진행한다면?**

**64bit윈도우 환경에서 8byte변수를 담은 구조체가 복사를 시도한다면 내부에서 바이트단위로 복사를 한다. 이는 Memcpy와 같은 형태로, 멀티스레딩 환경에서 Thread-safe하지않다.**

구조체 A,B가 있다고 가정한다.

**B = A;**

B는 A에 대해 대입받기 시작했고, 그 이후에 증감/차감등의 많은 변화를 거쳤다고 가정한다.

**A == B**

**그런데 대입이 다끝난후 두 값이 완전히 일치하는 경우도 생긴다.**

(흔치않은 상황이나 분명히 문제되는 상황이므로 파악하고 갈 것.)

**이는 바이트단위로 복사되기 때문에, 바뀐A의 값을 B가 그대로 복사받기 때문에 생기는 문제.**

((

추가적으로, AMD에서는 이러한 문제가 나야하는데 잘 나지않음.

AMD역시 구조체끼리의 대입에서 8byte변수는 byte단위로 Copy가 되는것을 확인했는데, 이상하게 테스트 되지않는다.

인텔쪽에서는 조금만 돌려도 바로 문제가 난다.

바이트 단위 복사를 최소단위 오퍼레이션으로 하는것은 불가능하다.

아직 파악하지 못한 부분.

))

**LockFree-Queue의 UniqueCount**

스택의 push에서 UniqueCount가 필요없었던이유는, ABA문제가 일어나도 노드를 추가하는데는 아무런 문제가 없었기 때문이다.

그렇다면 큐에 tail에는 UniqueCount를 쓰지않아도 상관없나? dd

이는 고민해볼것.

**<8.19>**

**멀티스레딩 디버깅 방법**

계속 언급하지만, 멀티스레딩 환경에서 printf나 파일쓰기와 같은 I/O작업이 들어가는 경우 스레드가 퀀텀타임을 포기하고 block상태로 진입하기 때문에, 문제가 나지않는다.

**보통은 여러 개의 스레드가 경합되면서 하면서 문제가 나기때문.**

로그를 굳이 화면이나 파일에 출력할필요 없다.

**굳이 자료구조가 아니더라도, 멀티스레드 환경에서 가장 간단한 구조로 순서대로 내역을 지켜 차곡차곡 저장만 되면 된다.**

(물론 이 디버깅데이터를 저장하기위해 Lock은 걸지 말아야 한다.)

이때 디버깅 데이터를 쌓기만 하는경우 포인터만 잔뜩 들어가게 되므로, 알아볼수있는 힌트거리를 추가적으로 더 넣어야 한다.

**추가정보로는 어떤행위인지 / 포인터 / 스레드ID 등이 있겠다.**

위의 정보를 바탕으로 역추적을 들어가야 하는데, 말그대로 역으로 추적하는 것이기 때문에 정확히 파악이 어렵다.

**이는 뾰족한 해결책이 없으므로 모든 데이터를 비교해나가야 한다.**

이는 서버프로그래머 라면 몸에 베어있어야 할 것.

(본인도 20만개 정도의 데이터를 모두 비교한 적이 있음)

지금 배우는입장에서 문제가 나면 좋은것으로, 손대지말고 추적해볼 것.

**LockFree Stack/Queue의 사용처**

사실 Stack이라는 자료구조는 메모리풀 이외에는 별로 쓸일이 없다.

**서버에서는 데이터 서순이 지켜져야 하므로 대부분의 자료구조가 큐가 된다.**

**LanServer의 Send링버퍼를 락프리 큐를 대체할 것이므로, Session개수만큼 큐가 생겨야 하므로 많은양의 큐가 생길것이며, 그외에 스레드단위 송신의 메시지 자료구조 역시 큐가 되어야 한다.**

**락프리스택 메모리풀의 통합**

우리는 Next포인터가 nullptr이 되는 것을 ABA문제로 삼았는데, 이것이 ABA문제인가?

LockFreeStack이 세개가 있다고 가정하자.

Stack 1

MemoryPool<NODE\*>

Stack 2

MemoryPool<NODE\*>

Stack 3

MemoryPool<NODE\*>

락프리스택마다 내부에 메모리풀이 존재한다.

**만약 Stack이 몇천개가 있다면 이는 낭비가 될것이므로, 메모리풀을 하나로 통일할수도 있겠음.**

(이는 당연히 static으로 간다. 템플릿안에 템플릿스태틱을 넣는게 문법적으로 까다로울 것)

**이 경우 메모리풀의 노드가 서로다른 스택에서 사용되고 반납되는 상황이 일어난다.**

**다른 스택에서 썼던 노드를 재사용한다고 해도 문제는 없을 것이다.**

**그리고 delete되지않으므로, next를 참조하다가 참조오류가 나는 상황도 없을 것**.

**문제는 해당 포인터를 받아 작업을 하거나 어떤 행위를 할 때 문제가 된다.(pop에서)**

pop에서, 들어있는 노드를 뽑아 데이터와 next를 확인하여 뽑아준다.

일단 확인하는 과정에서는 문제없을 것.

**뽑는 행위를 할 때, 우리는 ABA문제의 해결을 위해 별도의 Count를 두고 확인했다.**

따라서 우리가 알고있던 노드가 다른곳으로 할당되었다고 하더라도 문제될건 없어보이는데..

문제가 있을지 없을지 한번 고민을 해볼 것.

만약 성공한다면 꽤 효율적일 것.

**락프리스택 메모리풀의 통합 시뮬레이션**

[STACK1][A->(B)] A를 백업하고, B를 넣으려고함

이때 다른스레드에서 작업되어 STACK1::A가 반납되고, STACK2쪽으로 A가 alloc되고 push함.

[STACK2][A]

이 상황에서 A를 백업해놓고 있던 STACK1은 TopNode가 변경되었으므로(null) CAS에 실패.

**(스택의 CAS는 TopNODE의 노드를 백업노드와 비교하므로 다른쪽 스택인경우 실패 할 것.)**

**STACK은 내Top노드를 백업했다가 같은 일치하는 경우에만 PUSH한다.**

**-> 안전해보임**

**락프리 큐 메모리풀의 통합**

락프리 큐의 메모리풀 통합은 어떨지 살펴본다.

----------------------------------------------------------------------------------

[스레드1][Q1] A->B->(C) B백업, C를 alloc하여 Enqueue할 예정

[스레드3][Q1] 자료구조 변경시킴

[스레드1][Q1] D->E (현황)

[스레드2][**Q2**] 다른쪽 Q에서 B를 Alloc하고, next를 nullptr세팅.

----------------------------------------------------------------------------------

이때, 스레드1에서 멈췄던 작업을 진행

**B는 nullptr이므로 CAS성공. Q1데이터가 Q2쪽에 Enqueue되버림.**

**Stack (PUSH, POP) => TopNode의 Node를 비교하므로 ABA가 일어나도 문제없음.**

**큐의 Enqueue는 노드를 추가할때 tail.next == null을 비교하여 Enqueue한다. 따라서 내 큐에서 사라지고 다른큐에 할당된 노드라고 해도 Enqueue가 성공. => 자료구조 깨짐.**

이를 해결하고자한다면 CAS128로 바꿔야함.

---------------------------------------------------------------------------------

Enqueue의 작업

1. tail->pNext == nullptr 경우 Enqueue. (CAS)

2. tail을 tail->pNext로 바꿔준다. (DCAS)

---------------------------------------------------------------------------------

이때 큐의 메모리풀 통합을 위해 1의 과정에서도 DCAS로 바꾼다면 어떤가?

1의 작업을 DCAS로 바꾸자고 한다면

---------------------------------------------------------------------------------

**1. 백업한 노드가 일치**

**2. pNext가 nullptr이어야한다.**

---------------------------------------------------------------------------------

그런데 DCAS는 둘째치고, 백업할때 tail의 pNext값이 바뀌어버리면 답이없는 상황이다.

**내가 얻어오는 과정조차 안전하지않음**.

(스택은 백업노드를 보관했다가 TopNode->pNode와 비교한다. 따라서 UniqueCount로 체크하지않는 PUSH입장이라고 하더라도 문제될게 없음. (안에 자료(Data)가 바뀔수있지만, 이 데이터는 뭘로 바뀌든 노드껍데기가 같으므로 자료구조가 깨지지않는다. POP은 DCAS까지 하므로 문제될것이 없음.)

(큐의 Dequeue같은 경우 백업값을 얻어오는 도중 바뀌는경우라도 자료구조 자체가 바뀐것(DCAS)이므로 Dequeue하지않는다. 하지만 큐의 Enqueue 상황은 자료구조가 변경되지않았음에도 얻어오는 백업값(Tail)이 바뀌어버리므로 믿고 다음 CAS를 할수없는 환경이 되었음)

**tail의 next라고 얻었지만 걔는 이미 tail이 아닐수도 있음.**

**tail, top은 고정으로 존재하고 해당값을 수정하지만, tail의 next는 떠돌아다니는 노드.**

**따라서 이 방법은 애초에 불가능한 방법이므로, 큐마다 메모리풀을 가지고 간다.**

**Deque문제 - (락프리경험한문제.word)로 이동**

**락프리 vs 동기화 성능비교**

락프리 스택은 동기화객체를 사용하는 스택과,

락프리 큐는 동기화객체를 사용하는 큐와 비교가 되어야 할 것.

완성한다음 성능을 Test 해볼 것.

우리가 만든 락프리가 더 느리다고 하더라도 우리는 락프리를 사용할 것이다.

**Interlocked vs SRWLock**

단순히 인터락을 1회 호출하는것과, SRWLockExclusive를 Acquire-Release하는 것은 인터락이 더 빠를것이다.

유저 동기화객체 내부에서는 기본적으로 Interlocked를 활용하고 있다.

(비교대상이 유저동기화객체가 아닌 커널동기화객체인 경우 성능차이는 매우 심할 것)

(2)

---------------------------------------------------------------

**SRWLockExclusive();**

Data++;

Data++;

Data++;

Data++;

**SRWLockRelease();**

---------------------------------------------------------------

vs

---------------------------------------------------------------

InterlockedIncrement((long\*)&Data);

InterlockedIncrement((long\*)&Data);

InterlockedIncrement((long\*)&Data);

InterlockedIncrement((long\*)&Data);

---------------------------------------------------------------

**이 경우 SRWLock의 경우 캐시히트율이 100%이므로, 거의 대부분상황에서 더 빠르다.**

**그리고 위와 같은 상황이라면 Lock을 사용한다고 하더라도 거의 대부분 SpinLock과정에서 연산이 끝날 것이다. (퀀텀타임을 포기하지않는 선)**

**인터락 역시 경합이 발생된다면 성능이 하락될 것이며, 인터락이 많아질수록 성능저하는 더욱 크게 일어날 것이다.**

Interlocked < InterlockedExchange < CompareExchange < CompareExchange128

순으로 성능이 떨어질 것.

**Interlocked의 캐시히트 문제**

**인터락은 해당 값을 원자적으로 바꿔야 하는 작업을 해주는 역할이다.**

**CacheHit는 내 캐시메모리만을 바라보고 작업하기 때문에 성능향상을 볼수있는 것이다.**

**이때 인터락은 인자로 들어온 변수에 접근하는 모든 메모리를 차단하는 방식이다.**

**CPU차원에서 해당 메모리번지를 대상으로 값을 조회하려는 CPU를 막아버린다.**

**따라서 이 행동자체가 값을 대상으로 모든 코어의 캐시히트율을 없애버리는것이다.**

인터락이라는 행동자체가 값을 변경시키는것으로, 다른 코어에 있던 캐시들은 무효화된다.

[변경] 이라는 행위안에는 [읽기] 와 [쓰기]가 들어간다.

**위와 같은 SRWLock의 경우는 내 레지스터에 올려놓고 (++)라는 작업을 진행한다.**

**당연히 다른코어에 올라와있는 해당 메모리는 다 캐시무효화가 되겠지만, 내 코어에서는 캐시히트율이 100%로 진행된다.**

**반면 Interlocked같은 경우에는**

레지스터에 값을 올려다놓고 ++하는 행위와는 완전히 다르다.

**인터락 관련 함수들은 해당 포인터(메모리주소)를 꽂아넣고 [xadd 1]를 진행하는것으로, 해당 메모리를 잠궈버리고 1을 증가시키는 것이다.**

**LanServer에서의 인터락 vs 동기화객체**

위 상황은 경합을 일부러 만든 환경에서 테스트한 결과이고,

**우리가 지금 만드는 네트워크 라이브러리에서는 경합이 거의 발생하지않으므로 동기화객체와 인터락 중 어떤 것을 쓰더라도 큰 성능차이는 없다.**

**경합자체가 빈번하게 발생되지않는 구조는 동기화객체와 인터락의 성능차이는 크지않다.**

**애초에 워커스레드는 몇안되는것에 비해, 세션은 5천개 이상이 들어가기 때문에 두개이상의 스레드가 하나의 세션을 대상으로 깨어날일은 거의 가능성이 없기 때문이다.**

(가능성이 있는 부분은 Send완료통지/Recv완료통지가 같이 오는 것, 완료통지/컨텐츠쪽에서 어떤행동을 요청하는 것 정도가 될 것. 거의 가능성이 없다.)

**애플리케이션 입장에서의 Lockfree vs 동기화객체**

**애플리케이션 환경에 따라, 해당 스레드가 경합이 발생됐을 때 내 코어가 현재 스레드를 포기하고 Block상태로 들어가고 다른스레드의 일을 해주는 것이 더 효율적일 수도 있다.**

우리는 지금까지 스레드 동기화 방법을 고민을 할 때, 해당 스레드의 일을 어떻게하면 더 빨리끝낼수 있을 까를 고민했다.

**내가 지금할일을 CPU점유율을 계속사용하면서 루프를 돌린다면 10m/s안에 걸리고, 동기화가 걸리는 경우 20m/s가 걸렸다고 해도 이는 해당 스레드입장에서 이득일뿐,**

**애플리케이션입장에서는 어느것이 효율적인것인지는 알수없다.**

**<8.24>**

**Session 동시접근 문제**

IOCount는 Overlapped I/O가 잘못되는상황을 막기위해서 사용했다.

Lock의 경우는 Session을 사용중일 때 지워지는 것을 막기위해 사용했다. (이중 락)

(우리는 배열로 Session을 확보하고 가기 때문에 지워질일은 없을 것이다.)

**문제는 Session에 대해 Lock을 걸지 않을 것이기 떄문에, 바깥에서 별도의 업데이트 스레드나 컨텐츠 스레드에서 접근하여 어떤 행위를 할 수 있는 상황이다.**

**WorkerTheraed 안쪽에서는 특정 Session에 접근중일 때, IOCount를 물고있으므로 다른쪽에서 접근을 할수 없다. 따라서 문제가 나지않는다.**

**하지만 바깥에서는 Session에 대해서는 접근을 무조건 허용해주고 있는 상황이므로 내부에서 Session에 대해 Release를 진행하고 있는 상황이라도, 바깥의 요청으로 인해 동시에 접근이 가능한 상황이다.**

**상황 1. 바깥에서 접근한 Session이 이미 Release가 완료된 경우**

이 경우에는 어차피 보낼대상이 사라진 것이므로 상관이 없다.

SessionID는 유니크한 값이므로, Find되지않을 것이고 그대로 false를 리턴한다.

**상황 2. 바깥에서 접근한 Session이 재할당되어 다른 Session으로 바뀐경우**

유니크한 값인 SessionID로 Find할것이기 때문에, 찾지 못한다면 보통 false를 리턴할 것이다.

(바깥에서 요청하는 함수는 SendPacket, Disconnect 두개)

**하지만 검색이 끝나고 사용중에, 내부에서 해제-재할당 된다면 외부에서는 알 방법이 없다.**

결국 우리가 의도하지 않은 Session에 대해 무언가를 요청하는 형태가 되는 것.

----------------------------------------------------------------------------------

UpdateThread(contents thread)

**[SendPackeet요청 - FindSession - SendQ.Enque - SendPost]**

FindSession으로 Session을 찾아 로직 진행중..

...................................................................................

WorekrThread

**[Recv 0(연결 끊기 요청) -> if(IOCount-- == 0) ReleaseSession()]**

해제된 세션Index는 Index스택에 Push.

...................................................................................

AcceptThread

Accept() -> Index.pop -> Session[Index] 초기화 작업

(이 세션은 UpdateThread에서 사용중인 세션)

...................................................................................

SessionID로 Find하여 Session을 사용하는 도중에,

다른쪽에서 SessionRelease가 되어 재할당이 되는 경우

**배열의 경우 새로이 할당하는 것이 아니라 있는주소를 그대로 활용하는 것이기 때문에 이와 같이 즉시 Release하고 재할당되어도 코드내부에서 알수가없음**

----------------------------------------------------------------------------------

**Session 재할당 문제에 대한 해결**

Session에 대해 접근하여 사용할때, Release를 막아버린다면 근본적으로 해결될 것이다.

**ㄱ. Session에 Disconnect을 할 때 이미 Release가 진행됐거나 진행중이라면 빠짐**

**ㄴ. Session에 SendPacket을 할 때 Release를 타는경우 Release를 못하게 해야함.**

(ㄱ)의 경우는 단순히 내가 진행하던 Release구문을 빠져나가면 될것이나,

(ㄴ)의 경우는 Release구문을 타야하는데 바깥에서 사용함으로써 Release되지 못했으므로 내쪽에서 Release를 시켜줘야 한다.

**Session동시접근 차단?**

Session에 대한 Lock을 거는경우, 특정스레드에서만 Session이 사용이 가능하고 이 Session을 사용중일때는 아무도 접근하지 못한다.

또는 별도의 Flag를 두고서 한 스레드만 접근하도록 할 수도 있겠다.

**하지만 우리는 지금 누구든 접근하여 일을 할수 있게끔 의도하고 있기 때문에 하나의 스레드만이 접근하는 상황을 만들어서는 안된다.**

**지금 원하는 상황은 누구든 접근이 가능하지만,**

**한쪽에서 접근을 하는동안은 다른쪽에서 Release 되지 못하도록 막고 싶은 것이다.**

**Session에 대해 동시접근 하는경우 문제 (동기화이슈)**

하나의 Session을 대상으로 동시에 접근이 되는경우, 공유자원에 대한 문제가 생길 것이다.

Session동시 접근시 문제가 날 만한 것들을 살펴보자.

**1. Session - RecvQ링버퍼**

우리는 Recv완료통지에 대해 하나의 Overlapped구조체만을 사용하고 있기 때문에 Recv완료통지가 동시에 오는 경우는 존재하지 않는다.

(Recv를 1회로 제한하므로 문제가 나지않지만, Recv를 1회로 제한하는 것 과는 다른얘기.)

**2. Session - SendQ링버퍼**

SendQ는 동기화 문제가 존재하고있으므로, LockFreeQ로 대체하여 해결을 본다.

**Session에 대한 동시접근으로 인한 동기화 이슈문제는 위와같이 해결하고, Session이 동시에 접근하는 경우 재할당되는 문제를 해결하기위해 Release를 차단하는 문제로 넘어간다.**

**IOCount 기능의 확장으로 Release접근 차단**

접근을 허용하면서 Release만을 차단해야 하므로, IOCount를 사용한다.

**지금까지 우리는 I/O작업의 누적치를 IOCount로 두고, I/O작업시에는 Release되지않도록 했다.**

**이제부터는 IOCount의 기능을 확장하여 참조카운트 기능을 겸하도록 한다.**

**(Session에 대한 접근도 증가조건으로 넣어 Release를 막겠다는 의도이다.)**

**Session에 대해 접근하는 부분**

세션에 대해 접근하는 상황들로는,

**1. WorkerThread() GQCS완료통지 부분에서 접근**

**2. LanServer외부에서 SendPacket/Disconnect로 접근**

먼저, (1) 워커스레드 부분에서는 IOCount자체를 물고있기 때문에 고려대상이 아니다.

우리가 봐야할 것은 (2)의 외부접근.

**Session동시접근시 Release를 막기위한 해결방안**

Disconnect에서 접근한다고 가정하고 진행한다.

**Disconnect의 역할은 해당 Session의 연결을 끊고 Release되도록 유도하는 것이다.**

(Disconnect는 외부에서 끊기를 요청할 때 사용하는 함수이므로, 아직까지 호출해본적이 없다)

---------------------------------------------------------------------------------

Disconnect(세션ID)

{

Session = SessionFind(SessionID); (SessionID를 가지고 Array를 검색)

Session->Socket 끊기  
}

---------------------------------------------------------------------------------

이때 검색을 해서 찾았는데 해당 Session이 사라졌을 수도있고, 바뀌었을 수도있다.

위 동기화 이슈를 해결하기 위해서 Disconnect에서 할일은,

**1. Session을 사용할것이므로 Release를 막음 - IOCount증가**

**2. Release를 막기전에 이미 재할당 된경우 대비 - SessionID확인**

**3. 해당 Session에 대해 작업을 수행 (Disconnect) - Session작업수행**

**4. 막았던 Release를 할수있게끔 풀어줘야 한다. - IOCount감소**

**4-1. 여기서 사용하다가 Release 구문을 타지못했을 수 있으므로, IOCount를 차감했는데 0인경우 Release를 호출시켜줘야 한다.**

**하지만 여기서 문제는,**

**(1)의 경우(IOCount를 증가시키는 부분)에서 이미 Release가 될 Session이거나, 이미 IOCount가 0일 수 있다.**

**이 경우 IOCount가 혼자 0->1->0을 거치면서 Release가 두번될 수 있는 상황이다.**

**위 문제를 막기위한 추가 해결방안**

IOCount가 0이라는 것은 무조건 Release를 타게된다는 뜻이다.

그렇다면 if(InterlockedIncrement(IOCount) == 1)을 통해, 1인경우 중단하고 return하는것은 어떤가?

이는 ++한 IOCount가 1인 것을 확인하고 return하려는순간에 다른쪽에서 접근하여 ++된다면 2가되므로 구분하지 못하게 되므로 해결책이 될수없다.

**결국 차단하는 방식으로는 해결되지 않기때문에, 양쪽에서 서로 양보하는 모양새를 띄게된다.**

**만약 Release구문을 진입했는데 누군가 해당Session을 사용한다면 양보하고,**

**Session에 접근하여 사용하려고 했는데 Release될 Session이라면 포기한다.**

상당히 지저분한 방식이 되었다.

우리가 이전에 적용했던 이중락의 경우 리스트형태를 택했기 때문에 지저분했다.

**보통 세션관리자는 배열로 동기화없이 진행하고, 세션마다 락을 두는 것이 가장 일반적이고 깔끔한 방식이다.**

Release호출과 Release();

-----------------------------------------------------------------------------

If(IOCount-- == 0)

Release()

{

If(IOCount == 0 && ReleaseFlag == FASLS)

ReleaseFLag = true;

릴리즈진행.

}

-------------------------------------------------------------------------------

IOCount가 0일 때 ReleaseFlag를 최초로 TRUE로 만든 곳에서만 Release를 하게된다.

(이 작업은 당연히 CAS를 통해서 진행되어야 한다.)

**결국 IOCount로 Release진입 여부를 판단하고, Flag로 Release여부를 판단한다.**

따라서 이제 Release라는 함수자체를 확실하게 막았다.

IOCount차감시 항시 0으로 비교하여 Release호출하는 것도 유효하다.

**이제 Release내부에서 IOCount를 검사하기 때문에 언제호출하든 내부에서 막아줄 것이나,**

**불필요한 검사를 자주할 필요는 없으므로 IOCount가 차감되어 0이되는순간에 Release되게 한다.**

위와 같은 형태일 경우

내가 사용하고자 IOCount를 증감했는데 1이 나왔을 경우 바로 리턴**(Release)**해줘도 상관없다.

어떤 행위를 해도 상관없지만, 이미 끊겼기 때문에 해줄 이유가 없기 때문이다.

----------------------------------------------------------------------------------

**여러가지 상황 시뮬레이션**

추가적으로, 이제 어디서 사용하든 StartUseSession을 호출하므로 해제 자체를 막음.

StartUseSession전에 재할당 된경우, SessionID 비교로 잘못된 대상으로의 작업을 막음.

ReleaseFlag를 사용해 Release내부에서 중복을 막음

**1. 재할당**

**Find - StartUp - if(SessionID == Session->SessionID) { 할일 } - EndUp;**

**1 2 3**

다른스레드에서 종료요청이 오는경우(클라에서 종료)

**1. StartUp올리기 전이므로 종료됨. 재할당되어도 if문에 걸러짐**

**2. StartUp 이후이므로, 종료자체가 되지않음.**

**3. 2와 같음**

세션ID를 재확인하는 구문이없다면, 재할당되어 엉뚱한 세션을 대상으로 일을하는것을 막을 수 없음. (Send, Discon)

**A스레드 -> Find -> (StartUseSession호출 전) 이제 막 사용하려고함**

**B스레드 -> Recv 0(1) -> StartUse(2) -> Decrement(1) -> EndSession(0) -> Release**

**C스레드 -> Accept(재할당)**

**A스레드 -> 사용함(SendPacket.). 근데 ID가 안맞음. 리턴.**

---------------------------------------------------------------------------------

**2. 여러곳에서 종료요청이 오는경우 Decrement문제**

**A스레드 -> Find -> (StartUseSession호출 전) 이제 막 사용하려고함**

**B스레드 -> Disconnect()호출 (1 -> 0) 삭제.**

**A스레드 -> ID안맞음. return.**

**문제. Disconnect에서 Decrement하니까 동시에 두곳에서 하면 -1이된다. 그럼 Release가 안됨**

**A스레드 - Recv 0(1) -> Start(2) -> Decrement(1) -> End하려고했는데…**

**B스레드 - DisCon(1) -> StartUse(2) -> Decrement(1) - EndUse(0) Release하려했는데..**

**A스레드 - 하려던 End(-1) -> Release호출 (0이 아니므로 실패)**

**B스레드 - Release = 0이아님 실패**

**만약에 Release에서 -1일경우 그냥 Release해버리면?**

**해결방안. End에서 뭔가 조치. IOCount가 마이너스인경우, 0으로 바꿔주고 릴리즈호출.**

**어차피 둘다 0이하이므로 둘중하나는 ReleaseFlag에 의해팅기고 하나는 진행된다.**

**++추가**

ReleaseFlag만 확인하여 closesocket하면, 하나만 종료로직으로 들어가긴하겠지만 다른쪽에서 ++할수있는 상황이 생긴다. 이 경우 엉뚱한 클라를 끊어버릴 수 있다.

그래서결국엔 IOCount == 0, ReleaseFlag를 동시에 확인하여 종료구문을 타야한다.

그런데 문제는..

만약 -1이랑 0이 종료로직으로 왔다. 근데 0이 튕기고 -1이 들어가면 DCAS에서 문제가생김. 해제못함.

종료조건 DisconFlag로, IO가 다시 걸리는것을 막는다.

Recv가 걸려 블락인 상황에서는 IO를 감소시킬 방법이 없으므로, PQCS로 내가 종료통지를 보낸다. (이때만 예외로 -1을한다.)

이때 클라가보낸게 먼저오는 경우 Recv처리하고 IO를 더 안걸것(Discon)이므로 상관없음.

끊어야하는 상황에서는 무조건 Discon걸고 PQCS로 0보내버림.

우리종료처리가 먼저온다면 애초에 RecvIO를 걸지않으므로 작업하지않음.

종료로직을 타도록 해야한다. 이경우 어떤 IO도 걸리면안된다.

그러면 0이 떨어지게될것이다. 그런데 이때는 Recv가 걸려있는 상태에서 끊어야만 한다면..

Recv로 0을 PostQueuedComp하면 된다. 그럼 모든 로직이 끝이난다.

이때는 단순히 -1하는것으로

Start하고 Use하려는데 0이되었다. 그런데 Disconflag가 false라면..

1. IO작업은 Disconflag를 확인하고 들어간다.

2. Disconnect는 PQCS로 종료구문을 던진다.

3. ReleaseSession에서 DisconFlag가 FALSE인 상황이 있는가?

Start-Use에서 문제.. 내가 쓸려고올려놨음. 근데 다쓰고 내려야함.

이 내리는 순간.. 0이되었다고하자.

---------------------------------------------------------------------------------

**동시접근 시 성능**

**복잡한 구조로 코딩했지만 성능상 큰 이득은 없다.**

**우리는 현재 스레드보다 세션의 수가 월등하게 많기 때문에 경합이 발생할 일이 거의 없을 것이기 때문에, 성능상 큰 이득은 없을 것이다.**

굳이 이렇게 진행해보는 이유는, 멀티스레드에 대한 정확한 감을 익히기 위함이다.

StartUseSession / EndUseSession로직

// 이미 IOCount가 0인경우는 별도의 행위를 하지않는다.

if (IOCOUNT\_INCREMENT() == 1)

{

// ++의 행위로 Release를 못했을 수 있으므로 별도호출.

// 중복은 Release내부에서 막는다.

if (IOCOUNT\_DECREMENT() == 0)

ReleaseSession(pSession);

return false;

}

return true;

**재할당방지 로직후기**

Disconnect에 대해서는 알아서 구현하라고 언급했음.

그래서 나는 --IOCount를 하는것으로 간단하게 구현함.

이경우 언제든 IOCount가 -1이 될수있음.

Release에서 0보다 작은경우 0으로 돌려놓거나, IOCount가 0보다작으면 Cmp에 IC를 넣고 CAS를 한다고 하더라도 언제든 바깥에서 수가 변할수있음. 만약 수가변해서 못했다?

놉. 인터락이므로 어떤구문에서는 == 0은 타게된다.

근데 Release갔는데 -1되서 Release안될수도있는데?

그럼 0>= Release구문 타면 되는거아닌가?

(1) Recv 0왔음 -> IC 1에서 0으로만듬 0구문을 타려고드감

(2) Discon도 함.. (1) -> (0) -> (-1) -> (-2) 0으로만듬.

Recv 0왔음 -> IC 1에서 0만드려고함 만듬. DCAS감 Release함 0이므로 DCAS성공..

Discon왔음 2->1->0 -1됨 DCAS

**후기2**

애시당초 IOCount를 두는이유는 모든 IO가 끝난 경우 해제하기 위함이다.

만약 재할당되어 엉뚱한 완료통지가 오는경우 엉망진창을 막기위함.

그런데 IOCount를 끊어야한다고 --해버리면, 위 문제에 대해 자유롭지 못함.

따라서 일단은 I/O가 추가적으로 걸리는상황을 막고(DisconFlag), 이미 걸린 I/O를 완료통지를 유도하거나 끊어야한다(CancelI/O).

**Cancel I/O를 사용하는 경우**

소켓 INVALID로 변경, CancelIoEx 호출, DisconFlag 변경.

I/O 전후를 모두막았다.

간발의 차로 I/O가 걸리고 INVALID소켓이 되었따고 하더라도 CancelIoEx로 끊어줌.

**강제취소 없이 가는 방법**

Recv가 걸려있고, 상대로부터 응답이없다면.. 평생 못끊는다.

IOCount가 계속 0이므로.

컨텐츠 쪽에서 하트비트를 사용한다고 가정해보자.

n초간 응답이없어서 Discon을 강제로 끊으려고했다.

그런데 찰나에 응답이온다면? 그것도 종료처리가왔다. 이러면 결함임.

결국엔 Recv통지를 자연스럽게 유도해야하는데..

**<8.28>**

**(16기 수업 - <1.6> 추가분 합침)**

**ReadyQ에 대기중인 스레드가 없을 때**

타임 슬라이스가 모두 사용된 스레드가, ReadyQ에 대기중인 스레드가 없어서 계속해서 돌아갈 때 백업이 되는가?

컨텍스트 스위칭이 돌때는 백업이 된다.

하지만 이 경우 계속해서 세팅할 필요는 없으므로, 세팅은 다시하지않는다.

**Cancel I/O**

Cancel I/O는 걸려있는 비동기작업만을 취소하는것으로, 동기작업과는 관계가없다.

동기는 이미 리턴하는순간 완료가 된 것.

**완료통지가 오지않는 경우**

WSASend가 호출될 때, WSASend가 IOPENDING이 아닌 실패라면 완료통지는 오지않는다.

**static멤버를 전역에서 호출할 때**

WIN API수업에서 모니터링 클라이언트를 만들 때, winRroc를 클래스안에 넣기위해 winProc를 static멤버로 사용했다.

따라서 WinProc가 호출이 됐을 때 this포인터를 얻을 방법이 없게되었다.

**이를 해결하기 위해(this를 얻기위해) 핸들과 this포인터 테이블을 만드는 방법을 설명했고,**

**Extra라는 윈도우에서 별도로 제공된 공간을 활용을 했다.**

(GetWindowLongPtr() SetWindowLongPtr())

이는 윈도우마다 별도의 공간을 마련하여 해당공간을 사용하는 것으로, 윈도우핸들을 기준으로 구분할수 있게 되었다.

책에서는 이부분에 대해 먼저 설명하고있다.

TLS는 이와 비슷하다.

**TLS(Thread Local Storage)**

**동적TLS는 API차원에서의 기능**

**정적TLS는 컴파일러 차원에서의 기능**

**TLS는 OS차원에서 TLS기능을 제공해줘야 사용이 가능하다.**

**동적 TLS같은 경우는 이미 만들어져있고, 정적TLS는 요청하는 경우 만들어 준다.**

우리가 쓰는 런타임 라이브러리 역시 TLS기반으로 작동이 되고있다.

TLS를 사용한다면 동기화 같은 부분을 간단하게 해결할 수 있는 이점이 있다.

**Strtok 샘플과 CRL함수들**

Strtok()함수는 문자열을 지정하면 해당 토큰을 찾아주는 기능이다.

**Strtok는 일반적인 CRL함수와 달리, 그 전까지 찾았던 포지션포인터를 내부에 기억하고있다.**

따라서 Strtok()함수가 진행될때마다 계속 이어서 진행이 된다.

(일반적인 문자열 관련 함수인 경우 포지션을 계속 지정해야한다.)

**우리가 사용하는 런타임 라이브러리는 예전에는 thread-safe하지않았으나, window vista이후로는 모두 thread-safe를 보장하고있다.**

**따라서 어떤 변수를 선언하거나 버퍼를 제공하지않아도 내부에서 스레드별로 안전하게 기억하여 관리해준다.**

(이는 데이터영역(전역변수)가 아니며, 스레드가 늘어난다면 TLS도 늘어나게된다)

**Rand() 작동**

Srand()를 통해 인자를 전달하면 시드값이 저장되는데, 시드값은 스레드를 기준으로 저장된다.

따라서 멀티스레드환경에서 메인스레드만 srand()를 호출한다면 main스레드만 랜덤값으로 저장될 것이다.

Rand()함수 역시 TLS를 사용하고 있다.

**런타임 라이브러리 입장에서는 몇 개의 스레드가 만들어질지 알수 없으며, 그렇기 때문에 TLS를 활용하여 자동적으로 스레드마다의 고정적인 시드값을 세팅한다.**

**\_beginthreadex()**

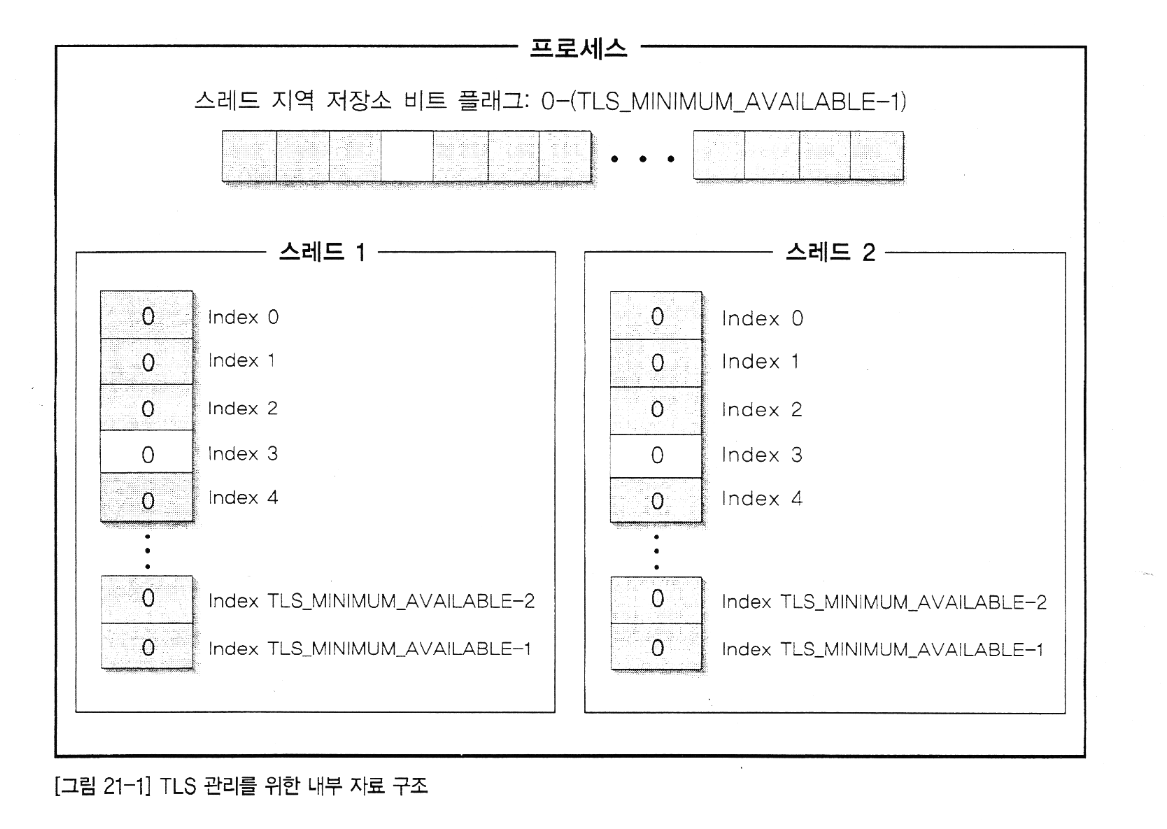
이전에 API함수인 CreateThread가 아니라, 런타임 라이브러리 함수인 \_beginthreadex()를 사용하라고 했었다.

**이는 CreateThread는 단순히 스레드를 생성하지만, \_beginthreadex는 내부에서 런타임 라이브러리가 준비하도록 작업을 해준다음 내부에서 CreateThread를 호출해주기 때문이었다.**

**TLS역시 사용하기 위해서는 런타임라이브러리에서 준비하는 과정이 필요하므로, 좋은 예시가 될수있겠다.**

이것 때문에 우리가 \_beginthread를 쓰자고 권장하는것이다.

**동적 TLS**



**동적TLS는 위와같이 각각의 스레드마다 이미 포인터 배열이 정해져있다.** (이미만들어져있다)

책에서는 최대개수가 64개로 설정되어있지만, 실제로는 훨씬 많이 존재하고있다.

(64개면 많아 보이지만, 런타임 라이브러리나 DLL등 여기저기서 TLS를 많이 활용하기 때문에 부족할 수 있다. 공간이 부족한경우, TLS에 포인터를 저장하고 포인터를 동적할당 하여 사용하는 방법도 있겠다.)

**동적TLS는 한계치가 존재하며, 한계치에 도달한경우 TLS할당이 실패하게된다.**

(한계치는 OS에 따라 결정된다.)

**동적TLS는 우리의 사용유무와 관계없이 처음부터 존재하며,**

**정적TLS는 요청하는만큼 용량이 그만큼 늘어나게된다.**

**TLS는 모든 스레드에 달라붙게 되므로, 마구 남발해서는 안될것이다.**

**TLS의 필요성**

우리가 전역변수를 선언하고 여러스레드에서 접근하게된다면, 공용으로 쓰는 하나의 공간으르 다같이 접근하는 꼴이된다.

이를 지역변수로 간다고하면 내 스레드 안에서만 활용을 하는 것이 된다.

(스레드마다 스택을 가지고 있기 때문)

**그렇다면 TLS를 사용하지않고 지역변수를 사용하면 되지않는가?**

**우리가 함수를 만들었는데 해당 함수가 호출될때마다 어떤 데이터를 계속 유지하고 싶은경우, 보통은 전역변수로 해결했다.**

**이때 스레드 내에서만 어떤 데이터를 유지하고 싶다면, 더 이상 전역변수나 static으로는 해결이 되지않을 것이다.**

이경우 스레드 함수에다가 변수를 넣어야 할것인데, 이런 방법을 사용한다면 스레드가 호출될때마다 포인터가 전달 되어야 할것이다.

**그리고 만약 이러한 기능이 언제 어디서 사용될지 모르는 클래스형태의 라이브러리일 경우 사용하는것이 불가능에 가깝다.**

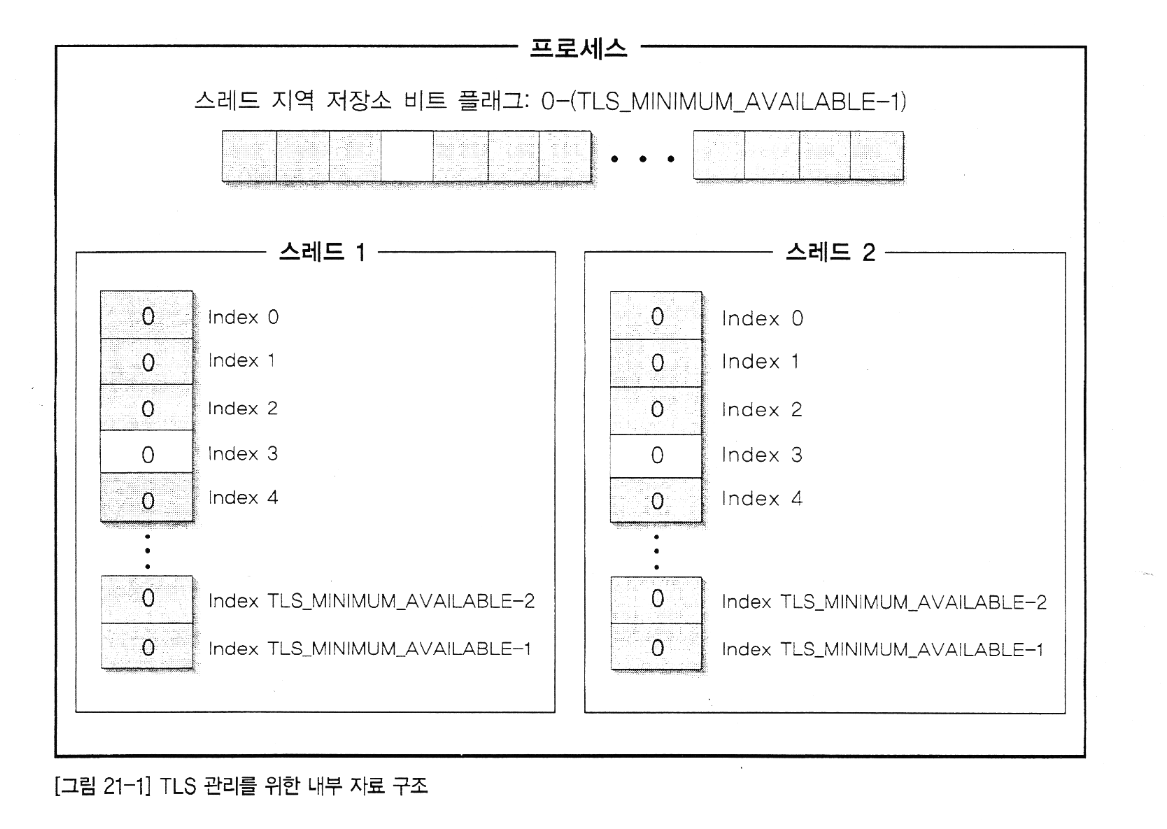
**따라서 TLS를 사용하는 것이 가장 적합할 것이다.**

TLS를 어떻게든 사용하지않고 지역변수로 해결을 할 수는 있기는 하지만, TLS를 사용하는경우 굉장히 간단하게 해결될 것이다.

**TLS내부**

만약 TLS내부의 5번 Index를 할당하고 사용하는경우, 같은 코드가 다른쪽스레드에서 호출됐을대 5번 Index공간을 사용할 것이다.

따라서 해당 Index는 내가 사용하는 스레드의 LocalStorage에 할당받아 사용해야한다.



**TLS의 할당은 프로세스차원에서 관리하며, Index번호 별로 사용유무에대한 비트플래그가 정해져있다.**

**TlsAlloc()**

DWORD TlsAlloc();

개념상으로 TlsAlloc은 사용자를위해 Index를 예약하는것과 같다.

**만약에 TlsAlloc이 Index 3을 반환했다면, 이 값은 TLS내에 수행중인 모든 스레드 뿐만 아니라 나중에 생성될 스레드까지도 접근할 수 있는 공간이 예약된다.**

**TlsAlloc은 프로세스 내에 비트플래그 배열에서 프리플래그의 위치를 찾아서 Index를 뱉어준다.**

TlsAlloc은 어떤 스레드에서 호출이되든 상관이 없다.

보통 클래스 안에있는 기능일 경우 생성자에서 호출하거나, main문에서 호출하게 될 것.

목적상 여러 개가 필요하다면 여러 개 확보해야 할것이다.

**그리고 해당 Index는 여러스레드에서 접근이 가능한 공간에 저장되어야한다.(head,전역)**

반환된 Index는 대부분 어플리케이션의 전역변수에 보관한다고 표현된다.

**보통은 클래스에 들어갈것이기 때문에 클래스의 멤버가 될 가능성이 가장크다.**

TlsAlloc이 실패한다면 TLS\_OUT\_OF\_INDEXES가 리턴된다.

**따로 할당실패를 알려주지않으므로, 우리 코드상에서 반드시 예외처리를 해줘야한다.**

**TLS 초기화**

**스레드가 생성되면 TLS는 해당스레드와 연계되는 수만큼 PVOID로 할당하고 0으로 초기화한다.**

**따라서 TlsAlloc을 통해 최초로 Index를 할당받아 접근한다면 해당값은 NULL일 것이다.**

이는 상당히 중요한 부분이다.

**TlsAlloc을통해 Index를 받는 것은 어떤 초기화함수나, 클래스에서 한번 호출하기 때문에 문제되지않는다. (보관해놓으면 끝)**

**그런데 문제는, 초기화하는 함수또는 클래스가 어떤스레드에서 호출될지 알수없다.**

**(스레드A,B가 있다고할때, 지금 실행되는 스레드가 A인지 B인지 알수가없음.(해당스레드에서 값을 할당했는지 안했는지 알 수 없음)**

TLS는 다른스레드에서 접근할 수 있는 방법이 없다.

**TLS에 접근할 것인데 데이터가 있는지 없는지 알수없으므로, TLS를 활용하는 코드에서는 항상 최초의 호출인지 판단하여 세팅을 하거나 초기화하는 작업이 무조건 들어가야한다.**

따라서 맨처음 이를 0으로 초기화 해줌으로서 최초할당에 대한 판단 기준을 만들어주는 것이다.

**추가적으로 사용하던 Index를 Free()이후, Alloc으로 재할당 한다면 0으로 안전하게 초기화시켜준다.**

이는 초기화 하는 시점에 모든스레드의 해당 인덱스를 0으로 밀어준다는 뜻이된다.

**사용 규칙에 대해서**

힙은 프로세스가 가동이되면 기본적으로 힙메모리 일정량을 commit해준다.

그럼에도 우리는 malloc을 통해서 힙메모리를 할당받아 사용한다.

스택 같은 경우도 다른스레드에 있는 스택메모리 포인터를 얻어다가 사용하는것도 가능하다.

**문법적으로 가능함에도 굳이 이렇게 쓰지 않는 이유는, 문법적으로 가능한 것들도 서로 규칙을 지키면서 안전하게 쓰기위함이다.**

TLS도 마찬가지로 Index를 받아서 사용하자는 규칙이 존재하는 것이다.

**TlsFree()**

BOOL TlsFree(DWORD dwTlsIndex);

Alloc받은 Index의 공간을 모두 사용했다면 TlsFree를 호출시켜 준다.

**TlsSetValue()**

BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);

이 함수는 스레드가 가지고있는 배열에서 Index 매개변수값을 이 value를 전달해서 세팅한다.

호출이 성공하는경우 TRUE를 반환한다.

단, 다른스레드의 배열에는 절대 접근하면 안된다.

**TlsGetValue()**

TlsSetValue로 할당한 값을 가져온다.

최초 리턴시 무조건 0을 리턴한다.

**TlsSetValue / TlsGetValue 사용시 주의점**

이 두함수는 성능을 위해, 내부적으로 에러확인을 해주지않는다.

**따라서 사용자가 엉뚱한 Index를 넣고 사용한다고 하더라도 내부에서 알려주지않는다.**

만약 런타임 라이브러리에서 쓰는 Index를 억지로 사용한다면, 모든 C/C++함수들이 오류가 날것이다.

**정적TLS**

정적 TLS는 변수하나를 TLS로 스레드라 선언하면 끝이다.

\_\_declspec(thread) static int TlsValue = 0;

declspec(thread)는 MS가 visual C 컴파일러에 추가한 한정자이다.

**(이 한정자는 전역변수 또는 정적변수로만 선언이 되어야 한다.)**

**위와같이 선언한 4byte공간은 이미 생성된 스레드, 앞으로 생성될 스레드, 또는 라이브러리에 만들 모든 스레드에 다같이 추가가 된다.**

**이는 컴파일러가 컴파일할 때 TLS라는 변수들을 포함시켜 링커과정을 고치는 것으로, 정적 TLS가 동작하기 위해서는 운영체제가 반드시 관여해야 한다는 뜻이된다.**

만약에 프로세스내에 새로운스레드가 추가적으로 만들어지면 당연히 OS입장에서는 이 스레드가 만들어질 때 이 정적 TLS를 사용할수있게끔 계속 추가적인 코드가 필요하다.

이는 DLL이 달라붙어 스레드가 추가된다고 해도 마찬가지이다.

모든스레드에 정적TLS가 추가되므로, DLL이 달라붙어 스레드를 생성한다고 하더라도 정적TLS가 생성된다.

이때 TLS가 어태치되는 순간 기존에 만들었던 스레드까지 정적TLS가 추가확보된다.

이는 매우 유용한 기능.

**정적TLS의 단점**

어플리케이션이 로드되고 시스템이 실행파일에서 Tls섹션을 발견하는 경우 모든 정적Tls를 저장할 수있는 메모리 블록을 할당한다.

**결국 컴파일러는 정적TLS변수를 참조하기 위한 추가적인 코드를 생성해야 한다.**

**이로인해 어플리케이션 크기가 조금더 커지고, 속도는 조금더 느려진다.**

(X86 CPU에서는 하나의 정적 TLS변수를 참조할때마다 세개의 기계어 명령어가 추가된다.)

**동적TLS vs 정적TLS**

**정적TLS는 추가적인 코드생성의 단점이 존재하지만,**

**동적TLS의 경우 함수호출이 일어나게 된다. (Alloc, Free, TlsSetValue, TlsGetValue)**

**따라서 성능에는 큰 차이가 없다.**

개인적인 견해로는 이미 만들어져있는 것을 사용하는게 좋다고 생각한다.

**정적TLS를 사용할때는 변수쓰듯이 쓰기 때문에 접두어로 표현하는 것을 권장한다.**

그래서 이개념을 활용하신다면 굉장히 유용하게 사용할수있다.

**TLS적용 - 락프리 메모리풀**

LanServer의 메모리풀이 TLS메모리풀이 되어야한다.

메모리풀을 아무리 락프리 스택으로 적용했다고 하더라도, 동기화 객체를 안썼을 뿐 성능이 향상된 것은 아니다.

이제는 어떤 동기화작업, 스핀락, 락프리 없이도 동시에 Alloc을 사용한다면 동시에 Alloc이 되어야 한다.

(이를 해결하기위한 방법으로는 TLS밖에 없다.)

TLS를 적용한다면 성능이 매우 올라가 갈 것.

**TLS적용 - 프로파일러**

지금까지만든 LanServer에는 프로파일러를 사용해볼만한 곳이 없다.

만약 락프리 메모리풀에 대한 Alloc을 성능측정하고 싶다면?

단순히 Alloc함수 위아래에 begin/end를 박아넣는다면, 여러스레드가 동시에 들어가므로 엉망진창이 될것이다.

**따라서 thread-safe하게끔 모든 TagName을 기준으로 프로파일링이 되게끔 해야하는데,**

락을 써서 이를 해결할 수는 없다.

**결국 프로파일링을 스레드별로 해야된다는 얘기가 된다.**

지금까지는 전역적인 공간에다 구조체를 두고 데이터를 저장했다.

**이제부터는 이 저장데이터들이 스레드마다 개별적으로 들어가야 한다.**

**이게 몇 개의 스레드든 사용법은 완전히 같으면서도 thread-safe하게 돌아야 할 것.**

PROFILE\_SAMPLE이라는 구조체자체가 배열로 나오고, 스레드마다 할당이 되어야 한다.

그리고 이제 begin-end의 호출은 소속된스레드의 TLS로 접근하는 것이다.

지금 저 같은 경우는 개별적으로 결과가 나오고있지만, 제대로 하고자한다면 같은 태그네임의 경우 최종적인 취합을 하는 것 좋을 수 도있다.

(굳이 합산해야될 필요성을 느끼지 못했기 때문. 이는 선택사항이다.)

**TLS프로파일러의 구현**

typedef struct PROFILE\_SAMPLE

{

long lFlag; // 프로파일의 사용 여부. (배열시에만)

WCHAR szName[64]; // 프로파일 샘플 이름.

LARGE\_INTEGER lStartTime; // 프로파일 샘플 실행 시간.

\_\_int64 iTotalTime; // 전체 사용시간 카운터 Time.

\_\_int64 iMin[2]; // 최소 사용시간 카운터 Time.

\_\_int64 iMax[2]; // 최대 사용시간 카운터 Time.

\_\_int64 iCall; // 누적 호출 횟수.

};

struct PROFILE\_THREAD

{

PROFILE\_SAMPLE Profile\_Sample[100]; // 같은 스레드내 측정할 샘플들.

HANDLE ThreadID;

};

PROFILE\_THREAD g\_ProfileThread[50]; // 성능측정 분류별

위와 같은 형식으로 가게될것이다.

**위와같이 따로 선언을 해놓고 하나하나의 포인터를 스레드마다 찔러줘야한다.**

**begin호출시 PROFILE\_SAMPLE을 동적할당해서 꽂는 경우도 생각할 수 있지만, 이 경우 본인것밖에 출력할 수 없게된다.**

우리는 어떤 스레드에서든지 출력을 원하는 스레드에서 일괄적으로 출력이 가능하게끔 해야한다.

**따라서** g\_ProfileThread**를 통해 관리당하게끔 따로 선언을 해놓는 것.**

**또한 각각의 스레드마다 TLS특정 인덱스에 포인터가 안전하게 들어가야 할 것이다.**

각각의 스레드마다 우리가 지정한 TLS특정 인덱스에 저 포인터가 하나씩 안전하게 들어간다.

위와 같은 배열형태를 취했다고한다면,

첫번째 begin을 하는 스레드에 Index 0번꺼 할당. GetTls**(GetAlloc)**

**프로파일러 - BEGIN**

결국 Begin이라는 애가 어떻게 만들어 져야하는가?

샘플을 얻어오는 코드만 바뀌면 나머지는 동일해도된다.

BEGIN("Tag")

{

**Sample = TagName검색** / 없는경우 할당

Sample->Start = QueryPerformanceCounter(……);  
}

이제 샘플을 검색하는 부분이 문제이다.

샘플포인터가 리턴되는 TagSearch()라는 함수가 있다고 한다면, 여기서 할일은?

이때 여기서 TLS인덱스는 미리 할당받아 놓은 상태이다.

(초기 프로파일러를 만들때도 프로파일러 전역에다 클래스를 하나 두고 생성자를 유도하여 초기화하는 코드로 들어갔다.)

**따라서 TlsGetValue()를 호출할 것인데, 만약 리턴값이 NULL인경우 최초 호출이라는 얘기이므로 Sample덩어리 데이터를 신규할당 해야할 것이다.**

이 Sample덩어리를 내가 쓸것만 Thread-safe하게 안전하게 뽑아내야 한다.

그리고 그 후에 TlsSetValue.

Sample = TagSearch(“Tag”)

{

P = tlsSetValue(Index);

If(p == NULL)

{  
 pt = PROFILE\_THREAD확보

tlsSetValue(index,pt);

p = pt;

}

Sample \*sp = p->Tag검색

Return sp;  
}

**여기서받은 p는 Sample이 아니다.**

**다시 내부에서 Tag를 검색하여 Sample포인터를 확보해야 한다.**

그리고 확보한 Sample포인터를 return하면 함수는 끝이난다.

**프로파일러 - END**

End도 마찬가로 Tag를 검색하는 파트에서 TlsGetvalue하여 그안에서 검색하게 될것이다.

END(“Tag”)

{

**Sample = TagName검색**

현재시간 = Query…

현재시간 = Sample->Start

Sample->Call++;

}

데이터를 저장하는 것은 기존의 프로파일러와 다르지않음.

그리고 정석적으로 본다면 소멸자에서 free를 하는 것이 맞겠지만,

**어차피 동적 TLS의 Index같은 경우는 프로세스가 꺼질때까지 유지되므로 free는 존재하지 않아도 상관없다.**

**ThreadID보관과 데이터취합**

위와 같은 형태로 구현하게된다면, 샘플링하는 과정에서는 PROFILE\_THREAD안에 ThreadID가 들어가지않아도 상관없다.

**단순히 ThreadID를 출력하기위해 넣은 것. (선택사항)**

그리고 최종적으로 모두 취합한 형태로 나오면 가장 이상적일 것이다.

하지만 취합하는 것 자체도 굉장히 번잡스러움.

이렇게하여 프로파일러를 완성한다.

**Tls에 대해 착각한 부분**

**Index를 통해 나온 값들은 스레드마다 다르게 보관한다.**

ThreadA, ThreadB에서 각기 TlsAlloc을하여, Index를 2,3을 받았다고 가정한다.

ThreadA Index = 2, SetValue 20.

ThreadB Index = 3, SetValue 30.

**이때, ThreadB에서 ThreadA의 Index인 2를 매개변수로 TlsGetValue를 하는경우 20이 나올거라고 생각했지만, 0(NULL)을 뱉어준다.**

TlsGetValue가 0을 리턴하는 경우는

1. TlsSetValue하지않고 최초로 TlsGetValue를 호출한경우,

2. TlsGetValue호출이 실패한경우.

하지만 위 상황에서 GetLastError를 호출하여도 0을 리턴하기 때문에 (성공적으로 수행됨) 이는 아직 아무도 세팅되지않았다는 뜻이된다.

따라서 TlsAlloc의 Index가 스레드를 공유하는 것은 아님.

**ThreadA의 Index가 10이라고 하여, ThreadB에서 TlsGetValue(10)을 하더라도 원하는 값을 얻지못한다.**

**TLS메모리풀**

MemoryPool<T>이 (1)

o-o-o-o-o-o-o-o-o-o

결국 메모리풀은 게임컨텐츠쪽에서 몬스터, 플레이어, 아이템 등 여러가지 오브젝트가 사용하게 될것이다.(템플릿 형태이므로)

**그리고 여러 스레드가 동시에 Alloc을 할 수 있게끔 Alloc의 시간을 최소화 시켜야 한다.**

**|Alloc| |Alloc| |Alloc| |Alloc|** (2)

Alloc을 여러스레드에서 접근하기 때문에 동기화/락프리/스핀락 등이 들어가야 한다.

**이때 동기화를 없애고자한다면 각자(스레드마다)의 메모리풀로 가는 방법 하나뿐이다. (Alloc마다 메모리풀)**

**이렇게 했을 때 문제는?**

**A스레드에서 Alloc한 것을 B스레드에서 Free하는 경우가 문제가 된다.**

(내가 Alloc한 스레드에서 내가 free를 하려면 결국 lock을 걸어야한다)

**각자의 메모리풀로 가는 것이 불가능한 이유**

A스레드에서 Alloc했는데, B스레드쪽에서 Free()해야 하는 상황으로는 무엇이 있는가?

**1**.

우리 LanServer는 에코서버에서 들어올 때,

LoginPacket은 AcceptThread내부의 OnClientJoin()함수에서 Send한다.

**따라서 AcceptThread에서 Alloc하고, WorkerThread에서 해제하는 형태가 된다.**

2.

우리는 현재 WorkertThread에서 OnRecv를 통해서 외부에다 알려주는 구조이다.

그리고 직렬화버퍼에 있는 데이터를 할당받아 외부 컨텐츠 파트에 알린다.

이 경우에는 WorkerThread() 내부에서 일어나는 일이기에 상관없겠지만,

채팅서버를 만들때만 하더라도 UpdateThread에 이를 직접 전달하는 형태로 만들수가 있다.

**이는 워커스레드에서 메시지를 받았을때 워커스레드 내부에서 직접적으로 로직을 처리하기 힘들때, 별도의 스레드를 만들어 일괄처리되게끔 던져주는 설계가 존재한다.** (예전에 언급함)

이때는 UpdateThread에서 뽑아서 사용하고 Free를 하는 형태가 된다.

**결국 Alloc()한 스레드와 Free하는 스레드가 나오기 때문에, 오로지 각자의 메모리풀로 가는 방법은 불가능.**

**Tls메모리풀의 구조**

하지만 메모리풀이 없어질 수는 없으므로, 두개의 개념을 섞는다.

**미리 일정량 Alloc하여 확보해놓고, 이 데이터들은 내가 따로 들고있는 형태로 간다.**

(반환역시 내 스레드에서)

**이때 일정량 Alloc할때는 동기화가 걸리겠지만, 이 데이터들을 스레드에서 가져가서 사용할때는 나혼자 쓰는 것이므로 여러스레드가 동시에 Alloc이 가능해진다.**

**이와 같은 Tls메모리풀형태의 단점**

**이와 같은 구조는n개의 데이터를 한번에 확보했는데, 이중 하나를 Free한다고 하여 메모리풀로 되돌릴 방법이 없기 때문에 메모리가 낭비되는 형태가 된다.**

**따라서 Free라는 행동은 다썼다고 알려주는 형태이며, 최종적으로 메모리풀로 데이터가 반환되는 시점은 n개를 모두 다쓰고 해제할 때가 될것이다.**

**Tls메모리풀의 청크데이터**

이때 n개의 데이터는 한번 Alloc시 재사용되지않으며, Alloc입장에서는 해당 청크를 다 썼다면 TLS메모리풀을 초기화하게 될것이다.

**만약 n개 데이터를 모두 소진했다면 새 청크를 받아 다시세팅한다.**

이때 할당받아 사용한 스레드 내부에서는 내가 할당한 n개의 데이터들이 어떻게 Free됐는지 알수없으며, 알 필요도 없다.

메모리풀을 청크하여 다시 스레드에 할당해주는 메모리풀의 입장에서는,

누가 어디서 쓰이는지 알고 있어야 하며 마지막인자가 반환되면 메모리풀로 청크데이터를 반환하게끔 구현해야 한다.

C에서 구현했던 Stack배열 방식으로 구현하면 될 것.

**Tls메모리풀의 최종적인 형태**

tls는 내 스레드만 접근이 가능하므로 이제 동기화의 걱정도 없어졌다.

**따라서 메모리풀을 포함하는 tls메모리풀은 n개의 덩어리객체를 관리하는 메모리풀이 될것이다.**

(보통 이를 청크라고 부른다)

이때 Free한 낱개의 객체들이, 다시 청크로 돌아간다고 한다면내 스레드가 아닌 다른스레드에서 Free를 한다면 결국 처음마주했던 문제와 직면하게된다.

(한쪽에서는 new를 계속하고, 다른쪽에서는 노드가 쌓이는 상황)

**따라서 최종적으로는 메모리풀로 돌아가되, 청크단위로 반납이 되어야 한다.(단순 Count증가)**

**메모리풀입장에서는 재사용이지만 청크된 노드는 재사용하지않는다.**

Free된 객체는 본인의 청크로 찾아 들어가야한다. 이는 알아서 구현할 것.

**이때 청크로 되돌아가는 객체는 굳이 주소를 돌려받을 필요가 없으며, 단순히 카운팅만 하여 청크 데이터크기만큼이 반납됐다면 메모리풀로 반납하면 된다.**

**어차피 메모리풀입장에서는 청크덩어리 단위로 Alloc하였고, 반납할때도 청크단위로 반납할것이기때문.**

**청크크기와 성능**

그러면 청크크기는 무조건 많이 잡을수록 성능이 올라가지않을까?

실제로는 비례하게 성능이 빨라지지는 않는다.

어느정도 까지는 효과를 보지만, 영향이없다.

왜 영향이 없는지도 알아내볼것.

((

예상 : 메모리풀TLS로 얻을 수 있는 성능은 결국 동기화를 없앴기 떄문에 얻을 수있다. 락프리역시 락을 없앴지만 동기화자체를 없앤 것은 아니다.(루프를 돌면서 재시도함)

동기화를 줄여서 얻는 성능에 더해 청크 크기가 크면 클수록 청크 덩어리의 Alloc/Free호출이 적어지긴 하겠지만 일정크기 이상이되면 Alloc/Free호출이 크게 차이나지않으므로 유의미한 차이는 없을 것.

))

**메모리풀의 형태 (List or Array)**

지금까지 만든 메모리풀 같은 경우에는 가변적인 길이, 락프리 구조등 여러가지 이유가 있겠다.

**추가로, 데이터를 Alloc하고 반환한다면 다시 자료구조에 넣어줘야 하는데 배열을 사용한다면 이를 빠르게 할 방법이 없기 때문이다. (더군다나 락프리는 배열로 구현이 불가능함)**

**하지만 청크의 경우 메모리풀에 데이터를 할당/반납을 반복하는 것이 아니라 정해진 크기를 뽑고 반납하기 때문에 굳이 리스트로 갈 이유가 없다.**

**따라서 기존의 메모리풀을 그대로 활용면서, 이를 한번더 래핑한 메모리풀TLS가 나올 것이다.**

(사용하는 메소드는 완전히 똑같이 가야한다. Alloc, Free.)

**TLS메모리풀의 최종적인 형태**

메모리풀 TLS클래스는 내부에 락프리 메모리풀을 가지는 형태로 간다.

**그리고 스레드에서 Alloc한다면 TLS클래스에서 청크를 받아 해당 데이터를 전달한다.**

**단순히 우리쪽에서 사용하는 클래스를 TLS메모리풀로 이름만 교체하는것으로, 스레드쪽에서는 일체의 수정을 하지않고 작업이 끝나야한다.**

이 형태가 완성된다면 new/delete보다 8에서 10배이상 빠르게 나와야하는 것이 검증되어야한다.

(우리의 프로파일러로)

**<9.14>**

컨텐츠쪽에서 많이 사용하는 Player라는 객체가 있다고 가정하자.

우리는 게임서버라는 클래스 안에 Player를 멤버로 넣고 사용할 것이다.

그리고 인스턴스 던전을 넣어야한다면 인스턴스 던전용 던전풀이 들어갈것이고, 해당 던전 내부에서 재사용되는 아이템이 있다면 해당 던전멤버 안에 아이템풀이 들어갈것이다.

위 방식은 우리가 지금까지 사용하던 방식이다.

(직렬화버퍼는 아님.. 직렬화버퍼는 사용에 제한을 두기위해 특이하게 만든 것.)

**TLS메모리풀**

**TLS메모리풀은 기존의 메모리풀과 인터페이스를 똑같이 가서 이름을 제외한 어떤 코드의 변경도 없이 바꾸도록 한다.**

**그리고 어디에서든 플레이어풀에서 플레이어를 Alloc받을 수 있어야하고, 어떤 스레드에서든 플레이어를 대상으로 Free를 한다면 Free되어야 한다.**

위와 같은 방식으로 접근하여 TLS메모리풀을 완성할 것.

(원래 있던 메모리풀 역시 보관할 것)

**청크하는 클래스가 결국 <T>를 다루는 템플릿 클래스 덩어리로 들고있는 형태이다.**

청크는 배열형태로, 결국 청크 덩어리를 어딘가에서 할당을 받아 써야하는 상황이다.

그리고 Free를 했을 때 어느청크로 들어가야 할지 알아야 한다.

ChunkAlloc(), ChunkFree()가 나올수도있으나, 성능을 생각한다면 없애는게 나을 것.

원래있던 메모리풀도 가지고있어야한다. 쓸곳이 있을것임.

**LockFreeQ내부 메모리풀의 TLS화**

락프리 큐/스택은 내부에서 메모리풀로 노드를 할당받아왔다.

이 메모리풀을 TLS로 대체할수있을까?

**동적TLS의 경우는 개수제한이 있기 때문에 사용하지못한다.**

**정적TLS의 경우 개수제한은 없을것이나 메모리가 많이 사용될 것이고, TLS변수자체가 전역변수가 되어야 하는 상황이므로 사용할 수 없다.**

따라서 락프리큐에 내장되어있는 메모리풀은 그대로 가야한다.

(TLS는 용도와 상황에 맞춰서 적용할 것.)

**실제 메모리풀TLS를 통한 객체 할당**

어딘가에서 메모리풀TLS를 통해 Alloc요청을 한다면?

if(TLSGetValue() == NULL)

{  
 새로할당  
}

else

{

청크를 받아 청크데이터를 SetValue();  
}

다음으로 청크데이터에서 객체를 alloc받아 return해준다.

**청크에서 객체를 받아 반환(Alloc)해줄 때, 먼저 Alloc이 가능한지 확인해야한다.**

따라서 별도의 AllocCount가 있어야하고, Alloc될수있는 크기도 정해야 한다.

**멤버로 MAXCount를 가지고있고 InterlockedIncrement()를 통해 동일해지는 경우 사용한다.**

MAX치로 Increment()한 Index가 MAXCount일 경우 해당 청크는 모두 사용한 것이므로, 마지막으로 사용한 곳에서 TLS를 NULL로 비운다.

따라서 그 다음으로 접근하는 스레드에서는 Alloc시 TLS가 NULL이므로 청크를 새로가져올 것.

이때 배열Index의 Count는 0에서 MAXCount까지 올라가게된다.

**Index감소방식 vs Index증가 방식**

보여주려고 했으나 안됨..

컴파일러마다 다른데 의도한대로 컴파일이 안됨.

**컴파일러 결과가 TRUE랑 FASLE랑 비교하는 것이 완전히 다른 경우가 있다.(컴파일러 결과가)**

어떤값과 비교하는 것은 무조건 Compare이다.

**그런데 TRUE/FALSE중 비교하는 것 중 하나가 비트연산으로 커버되는 경우가 있다.**

비트연산은 Compare가 아니므로 이게 훨씬 빠르다.

비교값이 변수라면 변수값까지 가져와서 Comp해야하고, 상수일경우 상수랑 Comp.

**TLS메모리풀의 성능**

TLS메모리풀은 아주 사소한것부터 영향이 가기 때문에 신경을 많이 써야한다.

**이 메모리풀은 결국 우리가 성능 테스트했을 때 8~10배정도의 성능향상이 있어야 한다.**

(new쪽에서는 생성자를 호출하고, TLS메모리풀에서는 생성자를 빼는 행위는 X. 사기임.)

이경우 정말 사소한 것 하나가 성능을 좌지우지 할것이다.

**코드단이 아닌 디스어셈블러 상태에서 어셈블리 코드를 보고 코드를 수정해나가야 할 것.**

멤버접근, 변수접근, 특정연산 어느것 하나라도 어떤 것이 나은지 한줄한줄 판단해야 한다.

모두 영향이 가기때문.

**성능향상 Free 예**

Free같은 경우도 Free하는 경우 FreeCount가 차감시키고 0, 또는 MAX수치가 됐을 때 해제한다.

그렇다면 InterlocekdIncrement, 또는 Decrment로 로직이 끝나게 된다.

청크의 MAXCount가 200이라면 199번은 InterlockedIncrement()를 시도하고 끝날 것이다.

그래도 여기서 성능을 줄일만한 부분이 있을 것이다.

**이러한 성능향상을 도모하는 이유**

단, 이렇게 높인 성능은 컨텐츠 로직에 비하면 미미한 수치이기 때문에 내 서버자체에 성능향상을 기대하기는 힘들다.

우리는 라이브러리를 만드는 입장으로 최대한 성능을 좋게 만들기 위해서 시도하고 있는 것이다.

실무에서도 이정도까지는 신경쓰지않는다.

우선은 코드를 보기쉽게 짜는 것이 최우선이며, 성능까지 부가되면 좋은 것.

Alloc도 결국 ArrayIndex를 하나씩 리턴할 것이냐, 아니면 list형태로 next로 찔러갈 것이냐 둘다 가능하다.

이부분에서 성능차이는 없음.

(?)--------------------------------------------------------

New/delete보다 10배 빠르다는 얘기는 따로얘기입니다.

New - alloc

Free - delete

-------------------------------------------------------------

**성능 테스트시 특이점**

테스트시 사이즈별(아마도 청크)로 다양하게 만들어 볼 것.\*

크기별로 다양하게. (200,300,500 등..)

포폴에 들어갈 것이므로 신경써서 만들 것.

**TLS메모리풀의 단점은?**

하나를 Alloc하더라도 청크하기 때문에 메모리낭비가 있을 수밖에 없다.

**TLS메모리풀 도입 후 모니터링의 변화**

확보된개수(Capacity), 실제 AllocCount가 있을 것이다.

낱개낱개에 대한 사용량을 카운팅하여 볼수 있겠지만, 이 방법은 멤버변수를 두고 Interlocked해야하기 때문에 성능이 많이 떨어진다. (동기화)

상황에 따라가되, 성능측정을 할때는 빠르게 가야할 것.

모니터링 작업 시 좀더 상세한 정보가 필요하다고 한다면 정교한 모니터링을 위해 이 방법을 사용해도 괜찮을 것.

Tls 완성요약

**메모리풀에서 예외처리 기능**

원래 우리의 메모리풀 역시 특정 풀에다가 엉뚱한 포인터를 Free로 넣었을 때 예외처리를 했다.

**서버에는 여러 개의 메모리풀이 들어갈것이고, 사용자가 PlayerPool에서 할당받은 포인터를 MonsterPool에 Free()의 인자로 넣어버릴 수있음.**

**위와 같은 에러상황은 사용자쪽 실수지만 Malloc/Free역시 사용자실수에대해 예외라도 던져주고 있기 때문에 우리도 이와 같은 기능을 추가한다.**

(에러를 방어하여 로직이 정상적으로 돌아가게끔 하자는 것은 불가능하다)

현재 로직에서는 이 같은 문제를 파악하지못하고있으므로, 파악하게끔 로직을 짤 것.

해당pool만의 기호를 박는 것은 너무 번거로운 방법.

**자동으로 넣게끔 하되 확인만 가능하면 되므로 간단하게 가면 된다.**

**단, 중요도는 성능이 가장 최우선이 되어야 할것이다.**