* 멀티 쓰레드 프로그래밍
  + Mutex 객체가 필요함
  + 성능차이 비교를 할 때는 Release 모드가 필수다
    - Visual studio가 너무 똑똑해서 루프를 다 돌지 않고 계산한다.
    - Volatile 붙이면 된다. 무조건 읽고 쓰게 강제한다.
    - 반대로 register을 사용하면 메모리에 저장이 안된다?
  + 느린 이유는?
    - 락을 얻지 못해 CPU오버헤드 발생, 큐 자료구조에서 빙글빙글돈다, 메모리는 하나인데 여러곳에서 찔러온다
    - 쓰레드가 많을수록 충돌이 많아져서 더 느려진다.
    - 싱글 스레드에서도 락있고 없고 차이에서 속도가 20배차이나난다 -> 락 자체의 오버헤드가 있다.
  + 해결 방법
    - 락을 쓰지않는다
    - 락없이 다른 thread가 실행되지 못하게 한다.
    - 어셈블리를 사용한다? (X) 32비트로 디버깅해야한다
    - C++11에서 atomic 연산을 할 수 있음
    - #include <atomic>을 사용
    - Sum = sum+2 는 atomic이아니다 읽고 쓴다 그사이에 다른 메소드가 끼어들 수 있음
    - Sum +=2 한번만 메소드를 호출하므로 atomic하게 돌아간다
    - Lock 이나 atomic연산을 최소화하는 방법으로 프로그램프 작성하는 것이 좋다.
      * Ex) lock을 반복문에서 쓰지 않는다. Local 변수 이용(로직을 변경)
    - 배열사용, 쓰레드 종료 후 합산
      * 문제 : 캐시 스레싱 문제 발생으로 속도가 느려짐
      * 배열 사이를 떨어뜨려놓으면 정상적인 값을 얻음
  + 요약
    - 멀티스레드 프로그래밍은 어렵다 : data race, 성능
    - C++11 표준 라이브러리를 사용
* 언리얼 엔진에서 멀티스레드
  + Game Thread : ai, 로직 등이돌아감
  + Render Thread
  + Audio Thread
  + 메인스레드가 3개이기 때문에 코어4개이상은 별차이가 없다.
* OpenGL ->Vulkan
  + 멀티스레드를 위해 OpenGL을 버렸다.
* D3D11 -> D3D12
  + 멀티코어에서 성능이 더 좋아짐(11에서는 멀티코어에서 성능이 좋아지지 않음)
* 멀티 스레드 프로그래밍 종류
  + Heterogeneous: 쓰레마다 역할이 다름, 구현이 쉽다는 장점, 쓰레드간 Load Balancing이 힘듬, 병렬성 제한, 성능 차이가 별로 없다?
  + Homogeneous: 모든 쓰레드가 같은 코드를 실행함, 놀고 있는 쓰레드 아무나 일을 바로 받아서 할수 있음 -> 로드 밸런싱 굿, 성능 제한요소가 없다 코어를 늘리면 늘릴수록 좋음, 병렬 자료구조가 필요함, MMORPG 게임서버에서 사용함
* MMORPG Game Server
  + 워커 스레드의 개수는 코어 개수1.5배
  + 멀티스레드 트릭: DataBase query
    - 쓰레드 blocking 초래
    - Blocking 전용 쓰레드를 따로두어서 작업 전달 시 해결가능
  + 멀티 스레드 트릭 : NPC AI
    - NPC가 너무 많음
    - 다른 스레드를 만들지 않고 해결해야함
  + Worst Case Tuning
    - 동접이 맥시멈이아닐 때 효율적으로 돌아가는 것은 의미가없다. 평상시 낭비는 괜춘
    - 최대 부하일 때 잘 돌아가는것이 중요
  + 컨테이너 자료구조 사용한 객체간 동기화
    - Ex) priority queue,set, queue 등
    - 멀티스레드에서 돌아가는 자료구조여야함 일반적인 stl vector는 성능저하가 크다
    - 여러가지 제약이 걸린 자료구조, 제약 조건을 이용해 최적화를 할 수 있음
    - 모든 제약조건을 갖춘 자료구조가 존재한다는 보장이없고, 직접, 커스텀 자료구조를 만들어야하는 경우가 있다
* Data Race 길들이기
  + 가정: Data Race를 피할 수 있도록 프로그래밍해보자 (나오는 값 예측가능, 락 사용하지 않으므로 성능도 좋을 것)
  + 실습#11
    - 릴리즈에서는 정상적으로 동작하지 않는다
    - false이기 대문에 멈춰야하는데 무시하고 넘어갔다.
    - while문이 컴파일이안됨, 컴파일러의 문제이다
  + 실습#12
    - Lock 사용 -> 해결은 되나 lock을 사용하지 않으려는 것이 목적이기 때문
    - Volatile 사용
    - 어셈블리를 알아야한다..
  + Volatile 사용법
    - Volatile int \* a: a가 가리키는값은 volatile이 적요되나지만 a는 적용이안됨
    - Int \* volatile a: a=b에 volatile적용이된다.
    - Volatile Qnode\* next (X)
    - Qnode\* volatile next (O)
    - 여러 개 쓰레드가 공유하는 변수는 volatile 사용해야함 그래야 컴파일러가 제대로 동작한다
    - Volatile이 만능은 아니다(CPU는 모른다)
* CPU에서 주의할 점
  + 상호배제
    - 여러 스레드가 동시에 실행될 수 없게 만들어야 한다.
    - 임계 영역(Critical Section)
    - 피터슨 알고리즘: 2개 쓰레드 사이의 락 언락을 구현하는 알고리즘, 3개 이상 에서는 동작하지 않는다, 매우 간단함
    - 락을 사용해아한다면 뮤텍스를 사용하는게 좋다.
* 메모리 일관성
  + 지금까지는 프로그램이 공유메모리에 대한 접근이 Atomic하다고 가정함
  + 근데 실제로는 그렇지 않다 상호 배제가 깨지는 결과
  + 피터슨에서 트루를 쓰고 읽어야하는데 먼저 읽어서 false가 읽혀서 그렇다.
  + 이 문제는 중단점으로 확인하기 힘들다 잠깐 나타났다가 사라지기 때문 -> 간접적으로 확인한다
  + Cpu 명령인 \_asm mfence를 사용한다(실행 순서를 바꾸는걸 막는다)
    - 64비트에서는 사용을 못함
    - atomic\_thread\_fence(std::memory\_order\_seq\_cst); 를 대신 사용
  + cpu가 atomic 하지 않은 이유는 사기를 치지않으면 실행속도가 느려지기 때문이다.
    - out of order execution 이기능을 끄는 옵션은 없다 너무 느려지기 때문
    - 또한 이것 때문에 오류난 적은없다.
    - 순차적으로 실행하는 척, HW에서 오동작을 커버한다, 싱글코어에서는 절대 안들킴
    - 내가 쓰고 내가 읽을때는 문제가 없다, 하지만 옆에 코어에서보면 순서가 어긋난게 보인다
  + Atomic<int>로 극복해보자
    - 느리다
  + 실습16,17
    - 두개의 캐시라인에 걸쳐져있는것을 bound가 가리킴
    - 메모리가 atomic하게 업데이트되지 않음 (0000 FFFF가 함께있다)
    - 캐시라인 사이즈는 무한대가아니기 때문에 발생한다
    - 포인터를 믿지마라, Byte밖에 믿을 수 없다
    - 이런 문제를 메모리 일관성 문제라고 한다
  + 해결 방향
    - 꼭 필요한 곳에  
      atomic\_thread\_fence() 사용
    - Atomic한 메모리 사용
  + C++ 11 에서 Atomic Memory
    - #include <atomic>  
       atomic<int> a
    - load(),store()를 사용해야하지만 불편하니까 =도 atomic하게 동작
    - 그냥 int 보다는 느리지만 뮤텍스 쓰는것보다는 빠르다
    - 종류
      * seq\_cst:   
        sequential consistency
      * Relaxed
      * acquire.release
  + Atomic Meomroy만 있으면 되는가?
    - No
    - 실제 프로그램은 다양한 자료구조를 사용함  
      (queue,stack,vector 등)
    - Stl 컨테이너는 모두 atomic하지않음 ->Crash 오류가 날날 것
* Non- Blocking 프로그래밍
  + 효율적인 구현
    - Lock 없는 구현
    - Lock 없으면 성능저하가 없나?
    - 상대방 쓰레드의 행동에 의존적이지 않는 구현방식이 필요함
  + 블러킹: 기다림
    - 성능저하
    - Priority inversion
    - Convoying: 락을 얻은 스레드가 일을 못하고 다른 스레드는 공회전함, 코어보다 많은 수의 thread를 생성하면 자주 발생
  + 넌블러킹: 안기다림
  + 넌블러킹 등급
    - 무대기(wait-free)
    - 무잠금(lock-free): 구현이 더 쉬움, 무대기와 달리 옆에 애와 충돌하여 밀릴 수 있다.
  + 멀티 스레드 프로그램에서 Non Blocking 자료 구조가 필요함
  + Atomic Memory를 만들어보자  
    wait-free를 유지하면서
    - 기존 싱글 스레드 자료구조도 atomic memory를 사용해서 멀티 스레드 자료구조로 변환은 불가능하다
    - Atomic Memory   
      외에CompareAndSet() 연산이 필요하다
    - CompareAndSet: 동기화 연산의 일종
  + 동기화 연산
    - 동기화 연산: 메모리에 대한 연산, 읽기, 쓰기도 동기화 연산임
    - CAS 추가가 필요
    - CAS(Compare And Set) 연산
      * Bool 메모리, 메모리 값이 expected 면 update로 바꾸고 true 리턴, expected가 아니면 false를 리턴
      * 조건부 write 명령
    - 모든 기존 자료구조를 wait free multithread로 변환 해주는 알고리즘이 존재함
    - CAS를 사용하면 모든 싱글스레드 알고리즘을 락프리 멀티스레드 알고리즘으로 변환 할 수 있으나 비효율적이다.
    - 락프리로 이미 만들어 놓은 것도 많다(Intel TBB, vs2015 PPL), 인터넷에서 가져다 쓴거는 검증이 필요함, 자신에게 딱 맞는 걸 만들자
  + 실제 CAS의 구현 (C++11)
    - Atomic compare and set대신 atomic compare exchange를 사용
  + Non Blocking 알고리즘 List
    - 중복 허용 X
    - 정렬되어 저장
    - 링크드 리스트로 구현( 삽입 삭제 O(1))
    - 메소드 : add remove contains
    - 성긴 동기화 (mutex사용) / Blocking임
    - 세밀한 동기화
      * 같은 노드가 아니라면 Lock을 할필요가 없다.
      * Add remove 시점의 prev curr 노드는 Locking 되야함
      * 노드 이동 시에도 lock하면서 이동해야함 , 이동하면서 unlock도 해야한다.
      * 너무 오래걸린다 -> locking을 너무 많이 하기 때문이다
      * 코어개수가 많을수록 빨라진다. 그러나 계속 빨라지지는 않는다
      * 결국 의미가 없다.
    - 낙천적 동기화
      * Locking 횟수를 줄이자
      * 이동시 잠금을 하지 않는다 -> 안하면 프로그램이 죽는다.
      * 크래시 : 제거된 노드가 tail을 가리키게 한다
      * 오동작: 제대로 잠겼는지 검사하고 잘못됬으면 처음부터 다시
      * Delete 를 하지않는다 (메모리 누수는 나중에 생각)
      * 유효 검사: remove가 됬는가? 끼어든 것이 없나?
      * 문제점: 기아를 겪을 수 있다. 그러나 매우 낮은 확률이기에 고려하지 않는다
      * Add,Remove전에 validate로 검사
      * Validate: 같은 키값을 찾을때까지 노드 이동, pred의 키값보다 큰값이 나오면 실패한 것이다.
    - 게으른 동기화
      * Validate가 처음부터 다시 순회하지않고 수행
      * Contains은 wait free로 만들자 -> marked를 이용하면 잠글 필요가 없다
      * 각 노드에 marked필드를 추가해서 제거됬는지 표시
      * Marking은 제거보다 먼저 수행할 것
      * 단점 : Blocking 알고리즘
      * 메모리 릭의 해결
        + Free List: delete 하지않고 모아놓는다(marking 해제시 오작동)
        + 언젠가 재사용해야한다
        + Shared\_ptr
      * Shared\_ptr
        + 스마트 포인터
        + Node next를 sptr로 구현해보자
        + 너무 느리다
    - Lock Free 알고리즘
      * 자료구조 및 그것에 대한 접근 방법
      * 멀티스레드에서 동시에 호출해도 정확한 결과를 만들어주는 알고리즘
        + STL 탈락, atomic한 동작
      * Non-blocking 알고리즘 (다른 스레드에 상관없이)
      * 다른 스레드와 충돌시 적어도 하나의 승자가 있으며 승자는 delay없이 완료된다
      * CAS가 핵심
  + Non Blocking 구현
    - 게으른 동기화에서 출발함
    - Lock() 사용 안함
    - 충돌 thread는 CAS로 승부
    - ADD
      * Marking이 false인가?(다른 스레드가 remove했는가)
      * Next가 Curr인가?( 다른 스레드가 Add했는가)
      * CAS(pred->making, false, false, pred->next, curr, new\_node) // 듀얼 카스는 x86 CPU에 없음
    - 동시 CAS의 극복
      * 한 장소에 주소와 Marking을 동시에 저장
      * 일종의 꼼수
      * 32비트 주소중 LSB를 마크로 사용
    - 더 이상 Next를 주소로 사용할수 없으므로 보조함수가 필요하다
    - Delete
      * 두 단계가 필요함 (curr 마킹, pred next = curr next / CAS사용)
      * 2번째 과정에서 실패할 수도 있다
      * 현실적 대안: 정책변경, 제거를 시도하지만 실패하면 그대로 놔두자
      * 모든 메소드에서 호출 시 마킹 노드를 제거하고 진행한다
    - 길이가 짧은 리스트(충돌이 자주 발생하는)에서 Lazy보다 성능이 좋다
  + EPOCH
    - 모든 공유 객체는 Epoch Counter 와 Thread Epoch Counter[쓰레드 개수]를 갖음
    - 일정 시간이 지나면 memory pool 객체를 delete함
    - 모든 스레들이 객첼를 보고있나 아닌가 -> 아무도없다 안전함
  + ㅇㅇ
    - Shared\_ptr의 atomic한 사용
      * 돈주고사거나 직접 구현
      * 기다린다(C++20)
    - Atomic\_shared\_ptr: blocking이다.
* **중간고사 범위 여기 까지**