논블로킹은 기존의 알고리즘으로 구현이 불가능하다. CAS가 추가되어야지 논블로킹 프로그래밍이 가능하다.

이 CAS가 무엇이고 어떻게 구현하는가? 에 대해 알아보자.

현실의 공유 메모리는 atomic하지 않다. 하지만 프로그래밍은 atomic memory를 가정해야 한다.

- **기본적인 질문**
    - 그러면 실제 메모리로 Atomic Memory를 제작할 수 있는 알고리즘이 존재하는가?
    - wait free를 유지하면서
    - 답 : 존재한다
        - 교재 참조
        - 이 강의에서는 자세한 설명 생략

int a; b = a; a = 3; 이렇게 쓰는 건 atomic이 아니다. 순서가 바뀌고 짠 대로 실행되지 않는다. 실제로 프로그래밍할 땐 atomic <int> a;라고 선언하고 프로그래밍을 한다.

Atomic memory는 어떻게 구현되느냐. lock 걸고 읽고 쓰면 당연히 atomic 한데 그럼 성능이 떨어지니까, wait free로 읽고 써야 겠다. wait free load store를 소프트웨어로 구현이 가능한가? 있다. 알고리즘으로 나와있다. 읽고 쓰는 순서가 엉망인 메모리들을 잘 달래가면서 atomic한 load와 store를 구현. 설명하는 시간이 너무 많이 걸리니 설명은 생략하고 대학원에서 배운다. int를 atomic <int>로 만들어주는 알고리즘이 있다.

- **기본적인 질문 2**
    - 기존의 싱글 스레드 자료구조도 Atomic Memory를 사용해서 멀티스레드 자료구조로 변환하는 것이 가능한가?
    - wait free를 유지하면서
    - 답 : 불가능하다
        - 증명은 교재에, 그리 어렵지 않음
        - 다다음주

그러면 우리가 사용하는 자료구조 map을 atomic 하게 바꾸는 것이 가능한가? 물론 atomic까진 쉽다. 모든 메시지 앞뒤로 붙이면 되니까. 근데 다른 스레드가 뭘 하는 거 기다리지 않고 동시에 실행되면서 돌아가는 그런 자료구조 구현이 가능하냐? 불가능하다. sort 알고리즘 중에 O(n) sort 알고리즘은 없다는 것이 증명이 되어있다. 그래서 우리가 이걸 만들기 위해 고생하는 일은 하지 않는다. 마찬가지이다. map. 교재에서는 queue가 예로 나왔는데 논블로킹 queue를 만들 수 있냐? 불가능이다. 왜일까? copy/ alu 연산 /goto 이런 걸로 논블로킹을 만드는 것이 불가능하다. 정말 불가능하냐? 이유는 다다음주에 배운다.

- **기본적인 질문 3**

    - 일반적인 자료구조를 멀티스레드 자료구조로 변환하려면 무엇이 더 필요한가?
    - wait free를 유지하면서
    - 답 : compareAndSet() 연산이면 충분하다
        - 증명 : 교재에 있음
        - 나중에 같이 증명

근데 불가능한데 왜 논블로킹 이야기를 꺼내냐. **기존에 배운 알고리즘 구성요소로 만드는 것은 불가능하다.** 하지만 여기에 + 알파를 더하면 가능하다. 이게 CAS 이다. **CAS가 추가되면 모든 자료구조를 wait free로 구현하는 것이 가능하다는 것이 증명되어있다.** 이것만 알면 **논블로킹 알고리즘을 자유롭게 만들 수 있다. CAS가 추가되면 wait free를 유지하면서 가능하다. Queue, vector, stack 등 모든 자료구조를 wait free로 구현하는 것이 가능하다.** 뜬금없이 나왔는데 이 cas , compare and set이 뭐냐? compare 비교 set 은 store랑 똑같은 단어인데 쓴다는 이야기다.

- **CAS (Compare And Set) 연산**
    - CAS (메모리, expected, update)
        - 메모리의 값이 expected면 update로 바꾸고, true 리턴
        - 메모리의 값이 expected가 아니면 false 리턴

```
if (*메모리 == expected)
{
	*메모리 = update;
	return true;
}
else
	return false;
```

이건 뭐냐 하면 atomic int라는 것을 구현한다고 하면 우리가 구현해야 하는 것은 store 하고 load를 메소드로 구현해야 한다. **bool CAS(메모리, expected, update).** value에 업데이트를 쓰는데, 쓰기만 하면 store랑 똑같은데 조건이 붙는다. value에 들어 있는 값이 expected 인지 비교를 한다. 같은 경우에만 upd ate를 한다. 쓰는데 조건부 대입이다. 쓰기 전에 값을 한번 확인하는 것, 이게 CAS이다.

메모리는 주소이고 **이 주소의 값이 expected랑 같으면 메모리에 update를 하고 리턴하고, 다르면 아무것도 안 하고 return 하는 것이 cas.** 뭐 대단한 건 줄 알았는데 별거 아닌데 문제는, 이 오퍼레이션이 돌아가는데 이게 atomic int다 한다면 모든 것이 atomic 하게 돌아가야한다. cas도 그렇다. 비교하고 쓰고 하는 이 동작이 atomic하게 실행이 되어서 **비교했다가 쓰는 동안에 그 사이에 다른 스레드가 절대로 끼어들지 않는 것이 보장이 되어있어야 한다. 그리고 보장을 하는 것이 어렵다.** mutex로 하면 당연히 보장되는데, 논블로킹 알고리즘을 구현하는 연산에다가 mutex를 걸면 논블로킹이 될 수 없다. 그러니까 이거 자체가 wait free로 구현이 되어야 하고, 그럴 수 있는 방법은 cas 뿐이다. 소프트웨어로 wait free 절대 구현할 수 없다. 그래서 cas는 하드웨어(기계어) 명령어로 wait free를 구현해야 한다.

copy, alu if-goto, 사실 copy는 alu에서 구현이 가능하다. a = b + 0 하면 b를 a에 카피하는 것과 동일하다. 그런데 alu와 if-goto는 cpu에 구현되어 있지 않으면 명령어를 사용할 수 없다. 마찬가지고 cas도 cpu에 있어야 한다. 우리가 사용하는 cpu x86에 있느냐? 있다. **지금 팔고 있는 모든 cpu에 다 있으니 어떻게 구현하나 걱정하지 않아도 된다.**

- **기본적인 질문 4**
    - 어떻게 CAS로 일반적인 자료구조를 멀티스레드 자료구조로 변환하는가?
    - wait free를 유지하면서
    - 답 : 알고리즘이 있다.
        - 모든 기존 자료구조를 wait free multithread로 변환해주는 알고리즘이 존재한다

```
class CASConsensus {
private:
	int FIRST = -1;
	AtomicInt r = AtomicInteger(FIRST);
public:
	value decide(value v) {
		propose(v);
		int i = thread_id();
		if (r.CAS(FIRST, i)) return
		proposed[i];
		else return proposed[r];
	}
}
```

어 그럼 cas만 있으면 된다고? 가능한데 방법을 모르면 못쓰는 게 아니냐??? 싱글 스레드 알고리즘이 지금까지 배운 모든 알고리즘을 멀티스레드에서 wait free로 구현하는 일반적인 방법이 있느냐? 있다. 우리가 만든 알고리즘 자료구조 클래스가 있고 여러 메소드가 있다. 데이터를 많이 갖고있는 클래스가 있고, 거기에 sort라는 메소드가 있어서 정렬이 되는 것이다. queue면 인큐, 디큐 메소드를 갖고 있는 것. 싱글스레드를 논블로킹으로 변환하는 알고리즘이 존재한다.

위 코드처럼 생겼다. consensus라는 클래스로 구현하고 그건 어렵지 않다. decide 메소드 하나밖에 없는 거고 cas를 호출한다. 나중에 다시 다룰 텐데 여러 스레드가 동시에 호출했을 때 어떤 값을 갖고 호출한다. 이때 어떤 값을 리턴하느냐 각각 요청한 값 중에 하나를 골라서 모든 스레드에게 그 값을 리턴하는 객체가 consensus라는 객체고 이 객체를 사용해서 밑에 유니버셜이라는 알고리즘을 사용할 수 있고 이게 변환 알고리즘이다.

```
class LFUniversal {
private:
	Node *head[N], Node tail;

public:

	LFUniversal() {
		tail.seq = 1;
		for (int i = 0; i < N; ++i) head[i] = &tail;
	}

	Response apply(Invocation invoc) {
		int i = Thread_id();
		Node prefer = Node(invoc);
	while (prefer.seq == 0) {
			Node *before = tail.max(head);
			Node *after = before->decideNext->decide(&prefer);
	   before->next = after;
	   after->seq = before->seq + 1;
	   head[i] = after;
		}
		SeqObject myObject;
		Node *current = tail.next;
		while (current != &prefer) {
			myObject.apply(current->invoc);
	   current = current->next;
		}
		return myObject.apply(current->invoc);
	}
};
```

이게 뭐냐? 우리가 만든 싱글 스레드 자료구조이다. 모든 자료구조는 클래스로 선언할 수 있다. 우리가 만든 자료구조가 있을 때 이 자료구조를 이 알고리즘으로 짠 거고 원래 이 알고리즘을 직접 호출해야 하는데 알고리즘을 호출하지 않고 한 겹 덧씌운 LFUniversal의 apply를 호출하면 이 알고리즘 하고 똑같이 호출하면서 멀티스레드에서 잘 돌아간다. 어떻게 CAS를 가지고 기존의 알고리즘을 wait free로 만드냐? 이렇게 쓰면 된다. 이 알고리즘이 정확하게 동작한다는 것이 이미 증명이 되어있다.

어 그러면 논블로킹 알고리즘 이런 거 있으면 좋은데 그럼 이야기가 너무 쉽다. 기존에 존재하는 모든 알고리즘을 lock free로 변환할 수 있다? 그럼 모든 게 다 해결된 것 아니냐? 멀티스레드에서 프로그램 문제점이 데이터 레이스, 컴파일러 cpu 캐시라인 문제가 있다. 그런데, 우리가 만든 알고리즘을 이렇게 한 꺼풀 덧씌우면 atomic 하고 lock free로 잘 돌아간다. 그럼 이렇게 하면 되는 거 아니냐? 된다. 정확하게 lock free로 돌아가고, lock free를 써놓은 이유는 wait free로 짜면 코드가 1.5배 더 길다. 심플하게 보이기 위해 lock free로 적어놨다. 문제는 성능의 문제가 있다.

- XEON, E5-4620, 2.2GHz, 4CPU (32 core)
- STL의 queue를 무 잠금, 무대기로 구현한 것과, CriticalSection으로 atomic 하게 만든 것의 성능 비교
  - test 조건 : 16384번 Enqueue, Dequeue (결과는 mili second)
  - EnterCriticaSection()을 사용한 것은 테스트 데이터의 크기가 100배
  - 따라서 100배 성능 차이 (4개 thread의 경우)
- 그렇다면 , EnterCriticalSection을 사용해야 하는가?
  - no, 멀티스레드에서의 성능 향상이 없다.

스레드 개수	1	2	4	8	16	32	64
무잠금 만능	3749	1966	1697	1120	742	525	413
무대기 만능	3640	1964	1219	1136	577	599	448
EnterCritical	232	822	1160	1765	1914	4803	7665

어떤 알고리즘을 lock free로 한다? 그럼 mutex로 구현하는 게 제일 쉬운데, 그럼 항상 그랬듯 멀티스레드로 돌렸을 때 성능 향상이 없다. mutex로 했으니 성능이 안나오고, 스레드가 늘어날수록 오버헤드의 이유로 점점 느려진다. 근데 이걸 안쓰고 lock free나 wait free로 구현하면 성능이 어떻게 나올까? 구현은 어렵지 않고, 쉽게 위에 나온 알고리즘을 쓰면 쉽게 할 수 있는데 성능이 어떻게 나오는지 돌려봤다. 위의 표가 그 측정이다. queue를 stl에 랜덤한 값을 인큐 디큐한 것을 atomic하게 한 걸 성능을 재보고, lock free wait free 버전을 비교해보니 lock free가 더 느리다. 막 붙여쓰니까 더 느리다 오히려.

멀티스레드에서 돌리면 성능 향상이 그래도 있지 않을까?? 싱글스레드에서는 느렸더라도. 그래서 스레드 개수를 늘려서 돌려보았다. 이게 가능한 이유가 벤치마크 프로그램을 돌린 곳이 32코어에서 돌려서 가능했다. 그래서 32 스레드가 될 때까지 성능향상이 일어났다. 그래서 결과적으로 32 스레드까지 가니까 mutex를 쓴 것보다 성능이 더 좋아졌다. 그래도 1개일 때 더 느린 게 오버헤드가 너무 크다. 쓰기 곤란하지 않나. 코어가 한 200개 정도 된다면 mutex보다 훨씬 빠른 테니까 그 정도면 써도 될까? 절대 아니다. 왜? mutex를 사용한 것은 테스트 데이터의 크기가 100배이다. 즉, mutex 스레드 1개일 때 나온 232ms는 100번이 아닌 1만 번 돌렸을 때의 시간이고, lock free는 100번 돌렸을 때 시간이다. 똑같은 횟수로 돌렸다고 하면 lock free wait free 칸에는 0을 2개씩 더 붙여야 한다. 32코어까지 간다고 해도 그래도 아직 속도 차이가 많이 난다.

위 알고리즘을 통해 mutex 없이 atomic 하게 돌릴 수 있었지만, 불가능하지 않았지만, 성능이 이렇게 안 나오면 어디에 쓰냐? 못쓴다. 너무 비효율적이기 때문에.

- 결론
  - cpu가 제공하는 CAS를 사용하면 모든 싱글 스레드 알고리즘을 Lock free 한 멀티스레드 알고리즘으로 변환할 수 있다.
- 현실
  - 비효율적이다
- 대안
  - 자료구조에 맞추어 최적화된 lock free 알고리즘을 **일일이 개발**해야 한다.
    - 멀티스레드 프로그램은 힘들다 -> 연봉이 높다
- 다른 데서 구해 쓸 수도 있다.
  - Intel TBB, VS2015 PPL
  - 인터넷
  - 하지만 범용적일수록 성능이 떨어진다. 자신에게 딱 맞는 것을 만드는 것이 좋다.

그렇다고 진짜 못쓰느냐? 그건 또 아니다. 최적화를 해주어야 한다. 아까 프로그램은 그냥 가능하다는 것을 보여준 거고, 반드시 그렇게 짜야한다는 건 아니다. 가능한 걸 알았으니 최적화를 해서 성능을 업그레이드해주어야 한다. 큐 스택 맵 벡터 트리 이런 걸 다 알고리즘에 맞춰서 따로따로 최적화를 해서 일일이 구현을 해야 한다. 그렇게 구현을 했더니 mutex보다 성능이 올라가더라. mutex보다 빠르다고 해서 끝나는 게 아니다. 락을 사용하지 않고 싱글 스레드를 쓴 것보다 빨라지더라. 이런 결과를 얻었다.

그래서 날로 먹지 못한다. 그래서 고생해서 최적화 알고리즘을 구현해야 한다. 알고리즘 별로 사용하도록. 이게 결론이다. 링크드 리스크를 논블로킹으로 구현하는데 2주를 쓸 거다. 8시간 동안 그것만 파야지 논블로킹 링크드 리스트를 만들 수 있다. 힘들다. 그래서 아무나 못한다.

꼭 개고생 해서 구현해야 하나? 아니다. 미리 구현해 놓은 것들이 라이브러리로 나와있다. 가져다가 쓸 수도 있다. 인텔의 TBB스레드 빌딩 블록이라고 논블로킹 알고리즘을 모아놓은 것이다. 인텔에서 만든 것이니 최적화도 잘 되어있다. 굉장히 신경 써서 잘 만들었다. 엄청 성능이 좋다. AMD CPU에서도 잘 돌아간다. 공식적으로 보증을 해주진 않는다. 경쟁회사 좋은 일 시켜주진 않지만 그래도 잘 돌아간다. 비주얼도 2015년부터 표준으로 추가시켰다. PPL이라는 걸. 이 안에 논블로킹 알고리즘들이 있다. 이걸 쓰면 잘 돌아가고, 부스트에도 있고, C++20에는 논블로킹 알고리즘들이 표준으로 또 들어갈 것이다 내년에.

또 인터넷에 여러 알고리즘 논문들이 나와있다. 간단한 건 공짜인데 좀 제대로 된 건 돈 받고 판다. 멀티스레드 프로그래밍이 힘들기 때문에. 구해서 쓸 수도 있다는 것이다. 근데, 프로그램을 만들 때 기존의 큐 스택 맵 이런 것만 쓰는 게 아니라 우리가 자료구조를 구현해서 쓴다. 근데 우리가 만든 알고리즘을 멀티스레드 논블로킹으로 돌리려면 어떡해야 하냐. 다른 곳에서 구할 수 없다. 없으니까 우리가 구현하는 거고, 우리가 만드는 자료구조가 싱글 스레드 버전도 없는데 멀티스레드용 버전이 있을 리가 없다. 그래서 직접 구현해야 한다.

- 정리
  - 성능 향상을 위해 멀티스레드 프로그래밍을 해야한다
    - 데이터레이스가 발생한다
    - 데이터레이스를 최소화 해야 한다.
    - 어쩔 수 없이 남은 데이터를 락없이 해결해야한다
      - 데이터레이스를 모두 없앨 수 없다
      - 락으로 해결하는 것은 성능 패널티가 크다
  - CAS를 사용하면 일반 자료구조를 멀티스레드 Lock Free 자료구조로 변환할 수 있다.
  - **효율적인 변환은 상당한 프로그래밍 노력을 필요로 한다. 알고리즘마다 따로따로 구현해야한다.**
    - 지금부터 경험해 볼 것이다.

성능 향상을 위해 멀티스레드 프로그래밍을 해야한다. 근데 성능향상의 이유가 아니라면 절대로 멀티스레드를 쓰면 안된다. 컴파일러 cpu 캐시가 방해를 하는데 그 방해를 뚫고 성능향상을 하는 게 어렵기 때문. 데이터 레이스라는 괴물과 싸워야 하는데 성능 문제도 없는데 굳이 괴물을 만들어 싸울 이유가 없다. 싱글 스레드 잘 짜서 만드는 게 훨씬 낫다.

데이터 레이스를 최소화해야 한다. 싱글 스레드를 멀티스레드로 옮기면 데이터 레이스가 여기저기 생기는데 그걸 잘 막아서 만들어야 한다.

- 실제 CAS의 구현 : C++11
    - Atomic_compare_and_set은 없고 atomic_cpmpare_exchange를 대신 사용

```
bool CAS(atomic_int *addr, int expected, int new_val)
{
	return atomic_compare_exchange_strong(addr, &expected, new_val);
}
```

```
bool CAS(volatile int *addr, int expected, int new_val)
{
	return atomic_compare_exchange_strong(reinterpret_cast<volatile atomic_int *>(addr),&expected, new_val);
}
```

논블로킹 알고리즘을 구현해야 하는데 atomic <int> a가 있을 때는 a의 load store cas라는 걸 호출해야 한다. 그럼 cas는 어떻게 구현해야 하는가? 얘는 alu if goto로 못한다. c++11에 있는 atomic_compare_exchange_strong이 있다. include atomic 하면 쓸 수가 있고, 이걸 호출하면 cpu에 내장된 기계어를 호출해서 연산이 실행된다. cas연산의 구현은 파라미터만 약간 바뀌고 호출하면 된다.

피터슨 알고리즘 했었을 때 모든 변수를 atomic으로 했더니 성능이 느려지더라. 동기화가 필요하지 않은 곳에서 읽고 쓰니까 파이프라인 스톨이 일어난다. 꼭 필요할 때만 동기화 연산을 사용하고 문제가 없을 때는 일반 오퍼레이션으로 해서 성능을 높이고 싶다. 그럼 volatile int 이걸 써야 한다. 그래서 파라미터로 atomic <int>가 들어가면 쓰면 되는데 volatile int a가 쓰고 싶다면 밑의 방식으로 해주어야 한다. 파라미터로 무조건 atomic 한 값이 들어가야 하고, atomic이 아니라면 캐스팅해서 넣어주어야 한다.

strong 말고 weak도 있는데 당연히 strong 쓰지 weak는 뭐냐. weak으로 해도 실행은 되는데 내부적으로는 strong으로 실행된다. arm cpu는 string이 없다. 그래서 addr에 써지고 값이 리턴된다는 보장이 없다. 모바일로는 그래서 strong을 못쓰냐? 아니다. arm에서는 계속 루프를 돈다. expected 값이 addr의 값이랑 다르다, 그럼 실패하고 같은 값인데 실패하면 성공할 때까지 계속 돈다. 그래서 strong을 구현한다. cpu마다 strong의 구현 방법이 다르기 때문에 우리는 그냥 strong만 하면 된다. arm도 내부적으로 strong이니까 그냥 쓰면 잘 돌아간다. weak를 써서 코딩할 일은 없다. 어차피 성공할 때까지 돌 거니까.

- **windows API**

-API

```
#include <windows.h>

LONG __cdecl InterlockedCompareExchange(
      __inout LONG volatile *Destination,
      __in LONG Exchange,
      __in LONG Comparand );
```

-CAS의 구현

```
Bool CAS(LONG volatile *Addr, LONG Old, LONG New)
{
      LONG temp = InterlockedCompareExchange(Addr, New, Old);
      return temp == Old;
}
```

옛날에는 이렇게 구현해서 했다. 이걸로 구현할 때 사람들이 버그를 엄청 양산했다. 앞이 기댓값이고 뒤가 쓰는 값인데 비교 값하고 위치가 바뀌었다. 그래서 CAS를 구현하게 되었다.

- LINUX API

```
#include <stdbool.h>

bool CAS(int *ptr, int oldval, int newval)
{
      return __sync_bool_compare_and_swap(ptr, oldval, newval);
}
```

리눅스는 이런 함수가 있고 이게 CAS의 역할을 해준다. 어쨌든 이런 건 C++11 이전의 이야기.

- **실제 HW (x86 계열 cpu) 구현**
  - lock prefix와 CMPXCHG 명령어로 구현
  - lock cmpxchg [a], b 기계어 명령으로 구현
    - eax에 비교 값, a에 주소, b에 넣을 값

```
if (eax == [a])
{
      ZF = true;
      [a] = b;
}
else
{
      ZF = false;
      eax = [a];
}
```

비교 값은 eax일 때 미리 넣어놓고 실행을 한다. 비교 값하고 메모리의 값하고 비교해서 같다면 zero flag를 true로 넣고 끝난다. 아니라면 실패를 하고 false를 flag에 넣는다. cas라면 여기서 끝나 지면 이건 exchange이기 때문에. 이 오퍼레이션을 atomic 하게 wait free로 cpu가 실행함. 그래서 cas 사용해서 프로그래밍했을 때 디스 어셈블로 따라가 보면 이런 명령이 나온다. 그래서 다음 주에 한번 쫓아가 보자.

- **실제 HW (ARM) 구현**

```
static inline AtomicWord CompareAndSwap(volatile AtomicWord* ptr,
      AtomicWord old_value, AtomicWord new_value)
{
      uint32_t old, tmp;
      __asm__ __volatile__("1: @ atomic cmpxchg\n"
            "mov %0, #0\n" "ldrex %1, [%2]\n" "teq %1, %3\n"
            "strexeq %0, %4, [%2]\n"
            "teq %0, #0\n" "bne 1b"
            : "=&r" (tmp), "=&r" (old)
            : "r" (ptr), "Ir" (old_value),
            "r" (new_value)
            : "cc");
      return old;
}
```

ARM에서는 어떻게 구현하냐 Compare and sum를 하나로 구현하지 않고 여러 개의 명령어로 구현한다. 이런 함수로 되어있다. 리눅스 소스코드에 있는 내용을 긁어온 것. 아까는 깔끔하게 cmpxchg 했는데 여기는 이상한 명령어로 되어있다. 링크밖에 안되기 때문에 계속 루프를 돌면서. 항상 성공하는 게 아니기 때문에 성공할 때까지 돌면서 해주어야 한다.

- 정리
  - 빠르고 정확한 병렬 프로그램을 작성하는 것은 어렵다.
    - 포기하라
  - 포기하는 것이 불가능하면 주의하여 프로그래밍하라
    - atomic 변수나 mfence의 도움이 필요하다.
  - 스레드 간의 동기화를 위한 자료구조가 필요하다
    - non blocking 자료구조가 필수이다
    - non blocking 자료구조에는 CAS가 필수이다.

빠르고 정확한 프로그램은 어렵다. 정확한 건 쉽다. mutex만 막 쓰면 되니까. 근데 빠르고 정확하기가 어렵다. 빠르기만 한 것도 어렵지 않다.

동접 8천 서버를 꼭 만들어야 한다. 그럼 어쩔 수 없으니 atomic 변수나 mfence 쓰던가 해야 한다. 근데 이런것만 사용해서 하면 성능하고 정확성을 둘 다 만족하는게 굉장히 어렵다. 적재적소에 놓아야 하고 남발하면 성능이 떨어지고. 그러니까 이런것 뿐만 아니라 논블로킹 알고리즘을 구현해야하고 스레드간의 서로 자료를 주고받는다, 동기화 한다, 커뮤니케이션 한다, 그럼 논블로킹 자료구조를 가지고 스레드 사이에 해야한다. 이걸 구현하기 위해서는 cas가 필수이다. 간단한 논블로킹 자료구조는 cas 없어도 된다. atomic int에 2를 더한다 이런 건 되지만, 큐나 스택 이상으로 복잡해지면 cas 없이는 논블로킹 알고리즘을 만들 수 없다.

다음 시간은 non blocking 자료구조를 맛보도록 하자. 링크드 리스트로 구현된 set을 볼 것이다. 그리고 atomic memory로 wait free 자료구조를 만들지 못함을 증명할 것이다.