목차

- 합의 객체
  - Non-blocking 알고리즘을 만들기 위해 필요한 객체
- 합의수
  - non-blocking 알고리즘을 만드는 능력
- 만능성
  - 모든 알고리즘을 멀티 쓰레드 무대기로

동기화

- 동기화
  - 자료구조의 동작을 Atomic하게 구현하는 것
  - 우리는 성긴/세밀한/낙천적인/게으른/비멈춤 동기화를 구현해 보았다.
- 동기화를 구현하기 위해서는 기본 동기화 연산들을 사용해야 한다.
  - 예) atomic_load(), atomic_store()
- 이 기본 동기화 연산들은 무대기(wait-free)혹은 무잠금(lock-free)이어야 한다.
  - 아니면 무대기나 무잠금 동기화를 구현할 수 없다.

합의(Consensus)

- 새로운 동기화 연산을 제공하는 가상의 객체
- 동기화 연산 : decide
  - 선언
    - `Ttype_t decide(Type_t value)`
  - 동작
    - n 개의 스레드가 decide를 호출한다.
    - 각각의 스레드는 한번 이하로만 호출한다.
    - Decide는 모든 호출에 대해 같은 값을 반환한다.
    - Decide가 반환하는 값은 전달된 value중 하나이다.
    - Atomic하고 Wait-Free로 동작한다.

합의(Consensus)

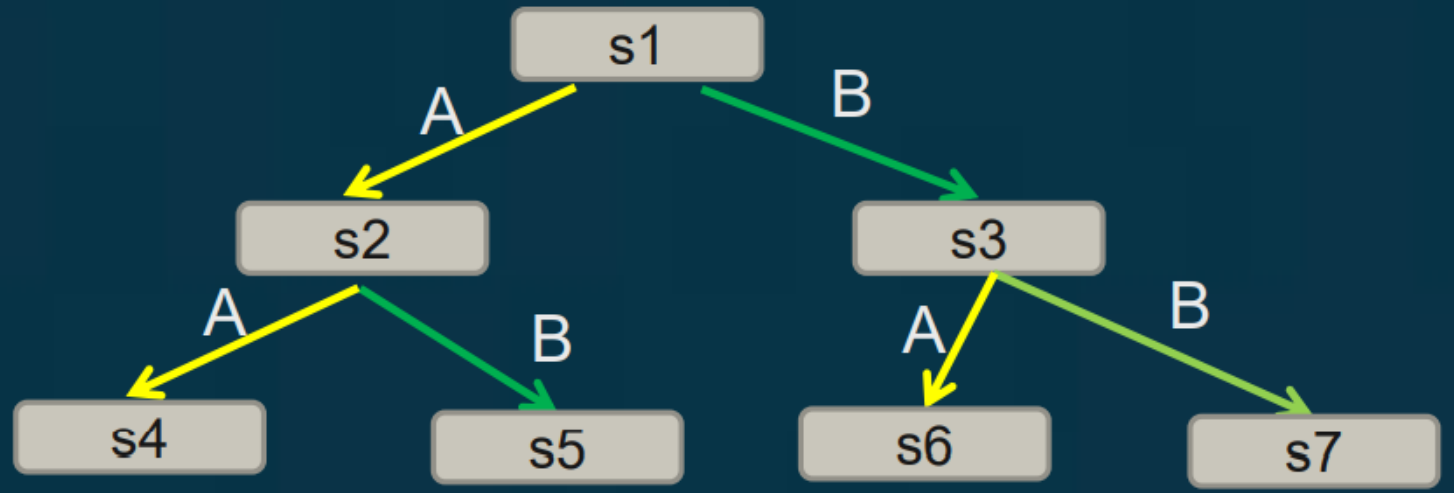
- 의미
  - 모든 쓰레드가 같은 결론을 얻는 방법
  - decide()가 모든 쓰레드가 wait-free로 같은 결론을 얻는다.
  - 여러 경쟁 쓰레드들 중 하나를 선택하고, 누가 선택되었는지 모든 쓰레드가 알게 한다.
    - 높은 확률로 제일 처음 decide()를 호출한 쓰레드가 선택됨

합의 수(Consensus number)

- 정의
  - 동기화 연산을 제공하는 클래스 C가 있을 때
  - 클래스 C와 atomic 메모리를 여러 개 사용해서 n개의 스레드에 대한 합의 객체를 구현 할 수 있다 => 클래스 C가 n-스레드 합의 문제를 해결한다(solve)고 한다.
  - 클래스 C의 합의 수(consensus number)
    - C를 이용해서 해결 가능한 n-스레드 합의 문제 중 최대의 n을 말한다. 만약 최대 n이 존재하지 않는다면, 그 클래스의 합의 수를 무한하다(infinite)고 한다.
  - 동기화 객체 C가 얼마나 파워풀 한가를 계측
    - 0, 1 : 있으나 마나
    - 2 : 2개 쓰레드 해결가능, 3개쓰레드 해결 불가능
    - 무한대 : 가장 파워풀한 객체

합의 수

- 문제 단순화
  - 알고리즘의 모든 실행 가능한 경로를 이진 트리로 나타낼 수 있다. 이를 프로토콜이라 부르자.
    - 프로토콜의 생김새는 구현 알고리즘과 input 값이 결정한다.
    - 왼쪽 : A가 이동, 오른쪽 : B가 이동
      - A, B는 read이거나 wrtie이다.
    - 실제 실행 시 어느 방향으로 진행할 지는 정해져 있지 않다.
      - 어느 방향으로 진행하더라도 올바른 결과가 나와야 한다.



합의 수

- 문제 단순화
  - 초기상태 : 아무런 이동이 없는 경우
  - 최종상태 : 모든 스레드들이 이동을 마친 상태 (프로토콜의 Leaf)
    - Decide 메소드가 결정된 값을 리턴한다.
- 일가(univalent) 상태
  - 앞으로 어떠한 이동을 하더라도 결정 값의 변화가 없는 경우
- 이가(bivalent)상태
  - 최종 결정 값이 결정되지 않은 상태
- 임계(critical)상태
  - 현재 상태가 이가이다.
  - 다음의 이동으로 무조건 일가 상태가 된다.
    - 두개의 child가 모두 일가 상태이다.

FIFO QUEUE

- 그래서?
  - Atomic 메모리의 합의수는 1
  - 2 Dequeuer QUEUE의 합의수는 적어도 2
  - 결론 : 2 Dequeuer QUEUE는 atomic 메모리로 구현 불가능
- 결론
  - atomic 메모리만 가지고는 큐, 스택, 우선순위 큐, 집합, 리스트등의 무대기 구현을 작성할 수 없다.

다중 대입 객체

- 그렇다면, n개 스레드의 합의 객체를 구현할 수 있는 합의수 n의 동기화 객체가 존재하는가?

- 존재한다.

- 다중 대입 객체 : 배열로 구성되며 복수의 원소를 atomic하게 변경할 수 있는 객체

동기화 연산들

- RMW (read-modify-write)연산
  - 하드웨어가 지원하는 동기화 연산의 한 종류
  - 특수 명령어가 반드시 필요 (wait-free를 위해)
- 메소드 M은 함수 f에 대한 RMW이다.
  - 메소드 M이 원자적으로 현재의 메모리의 값을 v에서 f(v)로 바꾸고 원래 값 v를 반환한다.

동기화 연산들

- RMW 연산의 종류 (원래 값이 x인 경우)
  - GetAndSet(v) : f(x) = v
  - GetAndIncrement : f(x) = x+1
  - GetAndAdd(k) : f(x, k) = x + k
  - compareAndSet(x, e, u)
    - f(x, e, u) = u(if x = e) , x(if x <> e)
    - 메모리의 값이 e면 u로 바꾼다. 리턴 값은 메모리의 원래 값
  - get()
    - 항등 함수 : f(x) = x
- 항등 함수가 아닌 함수를 지원할 때 그RMW를 명백하지 않은(nontrivial)이라고 한다.

동기화 연산들

- Common2 RMW연산
  - 많은 RMW연산이 여기에 속한다.
- 정의
  - 함수 집합 F는 모든 값 v와 F에 속하는 모든 함수 f_i와 f_j에 대해 다음이 성립하면 Common2라고 한다.
  - fi와 fj는 교환이 가능하거나 : $f_i(f_j(v)) = f_j(f_i(v))$
  - 한 함수가 다른 함수를 덮어쓰는 경우 : $f_i(f_j(v)) = f_i(v)$ 이거나 $f_j(f_i(v)) = f_j(v)$
- 특징
  - 합의수 2를 갖는다.
  - 최근의 프로세서들에서는 제거되는 추세이다.
- 예
  - getAndSet() : 덮어 쓴다.
  - getAndIncrement() : 교환이 가능하다.

합의의 의미

- 의의
  - 불가능한 시도를 미연에 방지할 수 있다.
    - 예) 원자적 메모리를 가지고 4개 스레드 무대기 병렬 큐를 작성하려 하는 행위
  - 구현 가능한 방법을 알고, 왜 그것이 구현 가능한지를 안다면 이를 최적화 할 때 더 잘 할 수 있다.

만능

- 모든 자료구조의 무대기 동기화가 가능 한가?
  - 그렇다! 가능하다.
- 만능 객체
  - 어떠한 객체든 무대기 병렬객체로 변환시켜 주는 객체
    - 예) 싱글 스레드에서만 돌아가는 큐를 무대기 병렬 큐로 변환 시켜 줄 수 있다.
  - n개의 스레드에서 동작하는 만능객체는 합의 수 n이상의 객체만 있으면 구현 가능하다.
    - 무한대의 합의 수 객체 CAS를 사용하면 쓰레드개수에 상관없이 만능 객체를 구현할 수 있다.

풀(POOL)

- 리스트는 Set객체
- Queue와 Stack은 Pool 객체
- Pool객체
  - 같은 아이템의 복수 존재를 허용
  - Contains()메소드를 항상 제공하지 않는다.
  - Get()과 Set()메소드를 제공한다.
  - 보통 생산자-소비자 문제의 버퍼로 사용된다.

풀(POOL)

- 풀의 종류
  - 길이제한
    - 있다 : 제한 큐
      - 구현하기 쉽다.
      - 생산자와 소비자의 간격을 제한한다.
    - 없다 : 무제한 큐
  - 메소드의 성질
    - 완전 (total) : 특정 조건을 기다릴 필요가 없을 때
      - 비어있는 풀에서 get() 할 때 실패 코드를 반환
    - 부분적(partial) : 특정 조건의 만족을 기다릴 때
      - 비어있는 풀에서 get() 할 때 다른 누군가가 Set() 할 때 까지 기다림
    - 동기적(synchronous)
      - 다른 스레드의 메소드 호출의 중첩을 필요로 할 때
      - 랑데부(rendezvous) 라고도 한다..

ABA

- 노드의 재사용시 생기는 문제
  - new(), free()는 메모리를 재사용한다.
  - CAS사용시 다른 스레드에서 그 주소를 재사용했을 가능성이 있다.

무제한 무잠금 스택

- 결론
  - CAS를 사용하여 Lock-free 스택을 구현하였다.
  - 메서드 호출은 스택의 top에 대해 성공한 CAS호출의 순서로 하나씩 진행되므로 순차병목현상이 나타날 수 있다.
- 메모리 문제
  - new와 delete를 사용하면. ABA문제가 생긴다.
  - Queue보다 문제가 생길 확률이 크다.

BACK OFF 스택

- CAS 동기화의 문제
  - 경쟁이 심할 경우 CAS 실패 시 계속 재시도하는 것은 전체 시스템에 악영향을 줌
    - Thread가 많아질 수록 경쟁이 심해짐
    - 경쟁이 심할 경우 CAS가 실패할 확률이 높음
      - 실패할 경우 성공할 때 까지 반복
    - CAS를 실행할 경우 같은 CPU의 모든 Core의 메모리 접근이 중단됨
      - Thread가 많아질 수록 잦은 메모리 접근 중단

소거

- 소거
  - 많은 쓰레드가 서로 충돌할 경우 Stack에 넣지 않고 직접 Data를 주고 받도록 한다.
    - Stack을 통하지 말고 다른 객체를 통해 전달
  - Lock-Free로 Data를 주고 받도록 한다.
  - 높은 경쟁률에 대비하여 주고 받는 별도의 객체를 복수로 준비한다.

소거

- 만약 push와 pop이 거의 동시에 실행된다면 두 연산은 서로 취소되어 없어지고 스택에 접근하지 않는다.
- 이런 경우 push를 호출하는 스레드는 스택의 변동 없이 pop을 호출하는 스레드와 값을 교환할 수 있다.
  - 이 때 서로를 소거(eliminate)하게 된다고 한다.

소개

- 일반적인 트리구조
  - 트리의 깊이를 유지하기 위해서 정기적인 재균형(Rebalancing) 작업이 필요(AVL트리..)
    - 하지만 병행 구조에서는 재균형작업이 병목이나 경쟁상태를 유발할 가능성이 있다.
- SkipList
  - 평균 O(logn) 검색시간을 갖는 병행 검색 구조
  - 재균형작업이 필요 없음!!!
  - 랜덤 자료구조(Randomized algorithm, Probabilistic data structure)
  - Worst Case O(n) 임

총정리

- 왜 멀티쓰레드 프로그래밍을 해야 하는가?
- 멀티쓰레드 프로그램은 멀티코어 CPU에서 어떻게 실행되는가?
- 멀티쓰레드 프로그래밍 API는 무엇인가?
- 내가 작성한 멀티쓰레드 프로그램이 왜 죽는가?
- Lock을 쓰면 왜 느린가?
- Lock을 제거했더니 왜 죽는가?

총정리

- Lock을 사용하지 않을 때 쓰레드간의 동기화는 어떻게 구현 하는 것이 좋은가?
- Non-Blocking 알고리즘이 무엇인가?
  - 어떻게 구현해야 하는가?
  - Blocking보다 왜 좋은가? 어떤 경우에 좋은가?
- Lock-Free 알고리즘의 구현이 왜 어려운가?

총정리

- OpenMP, TBB, CUDA, Transactional Memory는 무엇인가?
- 앞으로 우리는 어떠한 프로그래밍을 해야 하는가?
  - 언제까지 non-blocking 알고리즘을 써야 하나?
  - Core의 개수는 점점 늘어날 것인가?
  - C 스타일언어의 미래는?
  - 5년 후는?
  - 10년 후는?

정리

- 멀티쓰레드 프로그래밍은 피할 수 없다.
- 일반 프로그래밍에서는 볼 수 없는 많은 골치 아픈 문제가 있다.
- 여러 가지 방법으로 문제를 해결할 수 있다.
  - 성능에 주의 해야 한다.
- 일정 규모 이상의 멀티쓰레드 프로그램에서는 멀티쓰레드용 자료구조가 필요하다.
- Non-Blocking 멀티쓰레드 알고리즘의 사용이 필요하다.
- 자신의 요구에 딱 맞는 Custom한 병렬 알고리즘을 직접 작성해서 사용하는 것이 제일 성능이 좋다. <= 어렵다
- Transactional Memory, Functional Programming도 고려해보자.

강의 목적

- 멀티코어에서 성능을 올리려면?
  - non-blocking
- Non-Blocking은 얼마나 성능이 좋은가?
- 왜 어려운가?
- 얼마나 어려운가?
  - 프로그래머 요구사항, 비용
- 할 만한 가치가 있는가?