4. 동기화 연산과 CAS

멀티쓰레드 프로그래밍

정내훈

목차

- 병렬 프로그램 작성시 주의점
  - 컴파일러
  - CPU
    - 상호배제의 구현
    - 메모리 일관성 문제
- Non-Blocking 프로그래밍
- 이론 시간 + CAS

이론 시간

- 현실의 공유 메모리는 Atomic하지 않다.
- 하지만 프로그래밍은 Atomic Memory를 가정해야 한다.
  - 사람의 머리의 한계….
- 기본적인 질문
  - 그러면 실제 메모리로 Atomic Memory를 제작할 수 있는 알고리즘이 존재하는가???? .
  - wait-free를 유지하면서

이론 시간

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

이론 시간

- 기본적인 질문 2
  - 기존의 싱글쓰레드 자료구조도 Atomic Memory를 사용해서 멀티쓰레드 자료구조로 변환하는 것이 가능한가???? .
  - wait-free를 유지하면서
- 답 : 불가능하다.
  - 증명은 교재에, 그리 어렵지 않음.
  - 다 다음 주

이론 시간

- 기본적인 질문 3
  - 일반적인 자료구조를 멀티쓰레드 자료구조로 변환하려면 무엇이 더 필요한가??
  - wait-free를 유지하면서
- 답 : CompareAndSet() 연산이면 충분하다.
  - 증명 : 교재에 있음
  - 나중에 같이 증명

동기화 연산들

- 동기화 연산?
  - 다른 스레드와 통신하기 위한 기본 기능
    - 멀티쓰레드 알고리즘의 핵심
  - CPU의 명령어(또는 그 조합)으로 구현
  - Wait-free로 구현되어 있어야 한다.
    - 아니면 Non-Blocking 알고리즘에서 사용할 수 없다.
  - 기존 동기화 연산
    - Load (wait-free)
    - Store (wait-free)
    - Lock/Unlock (blocking)
    - atomic_thread_fence (wait-free)
  - CAS (wait-free)의 추가가 필요

동기화 연산들

- CAS(Compare And Set)연산
  - bool 메모리.CAS(expected, update)
    - 메모리의 값이 expected면 update로 바꾸고 true를 리턴
    - 메모리의 값이 expected가 아니면 false를 리턴

```
bool 메모리::CAS(expected, update)
{
    if (메모리.value == expected) {
        메모리.value = update;
        return true;
    } else return false;
}
```

    - atomic load/store로 구현 가능???
      - NO!!! (Wait-free 조건 때문)

이론 시간

- 기본적인 질문 4
  - 어떻게 CAS로 일반적인 자료구조를 멀티쓰레드 자료구조로 변환하는가??
  - wait-free를 유지하면서
- 답 : 알고리즘이 있다.
  - 모든 기존 자료구조를 Wait-free multithread로 변환해 주는 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];
            }
```

이론 시간

- 변환 알고리즘

```
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);
          } };
```

이론 시간

- XEON, E5-4620, 2.2GHz, 4CPU (32 core)
- Queue를 무잠금, 무대기로 구현한 것과, lock으로 atomic하게 만든 것의 성능 비교.
  - Test조건 : 16384번 Enqueue, Dequeue (결과는 milisecond)
  - lock을 사용한 것은 테스트 loop 횟수 100배
  - 따라서 100배 성능 차이 (4개 thread의 경우)

쓰레드 갯수	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
lock	232	822	1160	1765	1914	4803	7665

- 그렇다면, lock을 사용해야 하는가?
  - No : 멀티쓰레드에서의 성능향상이 없다.

이론 시간

- 결론
  - CPU가 제공하는 CAS를 사용하면 모든 싱글쓰레드 알고리즘을 Lock-free한 멀티쓰레드 알고리즘으로 변환할 수 있다.
- 현실
  - 비효율적이다.

이론 시간

- 대안
  - 자료구조에 맞추어 최적화된 lock-free알고리즘을 일일이 개발해야 한다.
    - 멀티쓰레드 프로그램은 힘들다. => 연봉이 높다.
- 다른 데서 구해 쓸 수도 있다.
  - Intel TBB, VS2015 PPL
  - 인터넷 (검증 필요)
  - 하지만 범용적일 수록 성능이 떨어진다. 자신에게 딱 맞는 것을 만드는 것이 좋다.

이론 시간

- 정리
  - 성능 향상을 위해 멀티쓰레드 프로그래밍을 해야 한다.
    - Data Race가 발생한다.
  - Data Race를 최소화 해야 한다.
    - Data race는 모든 오 동작의 근원
  - 어쩔 수 없이 남은 Data Race를 Lock 없이 해결해야 한다.
    - Data race를 모두 없앨 수 없다.
    - Lock으로 해결하는 것은 성능 페널티가 크다
  - Data Race는 공유 객체 때문에 발생한다.
    - 객체 : int, float, struct, class, container …
  - Non-Blocking 멀티스레드 객체가 필요하다.

이론 시간

- 정리
  - CAS를 사용하면 모든 일반 자료구조를 Multi-Thread Lock-Free 자료구조로 변환 할 수 있다.
  - 효율적인 변환은 상당한 프로그래밍 노력을 필요로 한다.
    - 지금부터 경험해 볼 것이다.

CAS

- 실제 CAS의 구현 : C++11
  - Atomic_compare_and_set은 없고 atomic_compare_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);
}
```

CAS

- 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;
  }
```

CAS

- LINUX API

```
#include <stdbool.h>

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

CAS

- 실습 #18 :
  - CAS를 사용하여 Lock()과 Unlock()을 구현한 후 1억 만들기 프로그램을 실행해 보자.
  - 1, 2, 4, 8개 쓰레드에서의 속도비교를 해보자.
  - 힌트
    - 0으로 초기화 되어 있는 공유 메모리 X가 있을 때. 모든 스레드에서 "CAS(&X,0,1)"을 실행 시키면 오직 하나의 스레드에서만 리턴값이 true가 된다.
  - 주의 사항
    - sum을 atomic <int>로 선언하지 말 것

CAS

- 실습 #18 : 예제

```
volatile int sum;
volatile int LOCK = 0;
void CAS_LOCK()
{
   /*  if (true == CAS(&LOCK, 0, 1)) */
}
void CAS_UNLOCK()
{
}
void worker(int num_threads)
{
   const int loop_count = 50000000 / num_threads;
   for (auto i = 0; i < loop_count; ++i) {
      CAS_LOCK();
      sum = sum + 2;
      CAS_UNLOCK();
   }
}
```

숙제 #2

- 숙제 2 : CAS lock의 구현
  - 제출물
    - .cpp 파일
    - 실행속도 비교표 (no Lock, mutex 사용, 빵집 알고리즘, CAS 사용)
    - CPU의 종류 (모델명, 코어 개수, 클럭)
    - 실행시간이 30분 이상 걸리거나 컴퓨터가 버벅거리면 속도 측정 생략 가능
      - 이러한 이상 현상이 생기는 원인에 대한 예측
  - 제출 : E-Class

CAS

- 실제 HW (x86 계열 CPU) 구현
  - LOCK prefix와 CMPXCHG 명령어로 구현
  - `lock cmpxchg [A], b` 기계어 명령으로 구현
    - eax에 비교값, A에 주소, b에 넣을 값

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

CAS

- 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;
 }
```

정리

- 빠르고 정확한 병렬 프로그램을 작성하는 것은 어렵다.
  - 포기하라.
- 포기하는 것이 불가능하면 주의해서 프로그래밍 하라.
  - atomic 변수나 mfence의 도움 필요
- 쓰레드 간의 동기화를 위한 자료구조가 필요하다.
  - Non-Blocking자료구조가 필수이다.
  - Non-Blocking자료구조에는 CAS가 필수이다.

다음 시간

- Non_blocking 자료구조 맛보기
  - 링크드 리스트로 구현된 Set
- atomic memory로 wait-free 자료구조를 만들지 못함을 증명

질문???