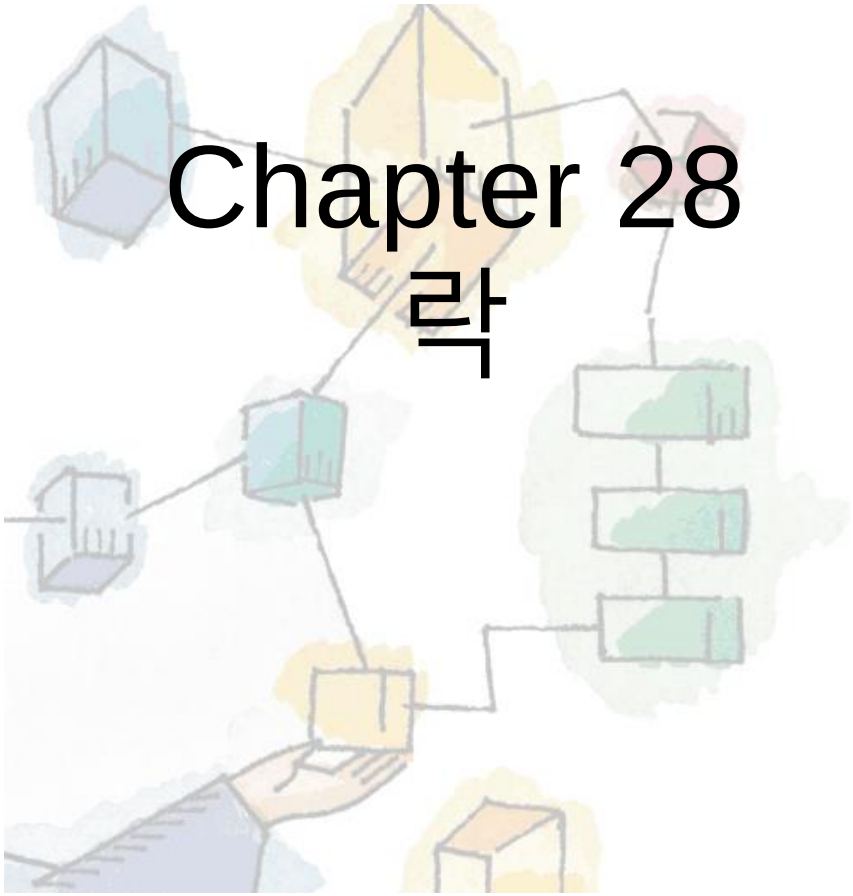
Chapter-28

운영 체제

정내훈

2023년 가을학기
게임공학과
한국공학대학교

Chapter 28
락



락 : 기본 개념

- **임계 영역**을 마치 하나의 원자적 명령어인 것처럼 실행하게 한다.
  - 예 : 공유 변수의 일반적인 수정

```
balance = balance + 1;
```

  - 임계 영역 주변에 락 추가

```
1    mutex mylock; // some globally-allocated lock 'mylock'
2    …
3    mylock.lock();
4    balance = balance + 1;
5    mylock.unlock();
```

락: 기본 개념

- 락 객체는 락의 상태를 저장한다.
  - **사용 가능 (available,** 해제, **unlocked** 또는 **free**)
    - 락을 획득한 쓰레드가 없다.
  - **사용 중 (acquired** 또는 잠김, **locked**, **소유 중**)
    - 어느 **한** 쓰레드가 락을 가지고 있고, 임계 영역을 실행 중이다.

lock()의 의미

- `mutex::lock()`
  - 락의 획득을 **시도한다.**
  - 아무도 락을 갖고 있지 않으면, 락을 **획득**한다.
  - *임계영역에* **진입한다.**
    - 이 쓰레드가 락을 소유했다 라고 한다.
  - 다른 쓰레드들은 이 쓰레드가 락을 갖고 있을 동안에는 임계 영역의 진입이 막혀서 *뭠춰있게 된다.*

C++11 락 - mutex

- C++11에서의 기본 락 클래스.
  - 쓰레드 사이의 상호 배제를 구현하기 위해 사용.

```
1    mutex mylock;
2
3    mylock.lock();
4    balance = balance + 1;
5    mylock.unlock();
```

  - 다른 변수(balance 말고 다른)를 보호 해야 한다면 다른 락(mylock 말고 다른)을 사용해야 한다 -> 병행성(concurrency) 증가 (보다 세밀한[fine-grained] 접근법)

락의 구현

- 효율적인 락은 low cost.로 구현해야 한다.
- 그러한 락의 구현은 **하드웨어**나 **OS**의 도움이 필요하다.
  - 적은 개수의 명령어로 구현 => HW
  - Lock을 얻지 못했을 때 쓰레드를 대기 상태로 => OS

락의 평가 – 기본 목표

- **상호 배제**
  - 기본 사항, 여러 개의 쓰레드가 임계 영역에 동시 진입하는 것을 막는다.
- **공정성**
  - 락 획득의 기회가 공평하게 주어지는가? (기아 문제, Starvation)
- **성능**
  - 락 자체의 구현에 오버헤드가 적어야 한다.

구현 : 인터럽트 제어

- **싱글 CPU, 싱글 코어**에서만 가능한 락의 구현법
  - 인터럽트를 금지 시킨다.
  - 초창기 해법

```
1    void lock() {
2        DisableInterrupts();
3    }
4    void unlock() {
5        EnableInterrupts();
6    }
```

  - 문제:
    - 응용 프로그램에게 너무 큰 권한을 준다.
      - 악용의 여지가 있다.
      - 버그로 인한 오동작의 여파가 크다.
    - 멀티 CPU, 멀티 코어에서 동작하지 않는다.
    - 인터럽트의 중지는 I/O 오동작의 위험이 있다.
    - 현대 CPU에서 인터럽트 컨트롤은 느리게 실행된다.

왜 하드웨어의 도움이 필요한가?

- **첫 시도**: flag을 사용해서 락의 획득 여부를 판단하자.
  - 문제 있는 코드

```
class mutex {
   bool flag;
public:
   mutex() { flag = false; }
   void lock()
   {
      while (true == flag);
      flag = true;
   }
   void unlock()
   {
      flag = false;
   }
}
```

왜 HW의 도움이 필요한가? (Cont.)

- **문제 1**: 상호 배제 실패

Thread1	Thread2
call `lock()`	
`while (flag == true)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (flag == true)`
	`flag = true;`
	interrupt: switch to Thread 1
`flag = true;` // false를 한번더!!!	

- **문제 2**: 회전 대기(Spin-waiting) :
  - CPU시간 낭비. 다른 쓰레드가 유용하게 사용할 수도 있는 CPU시간을 아무런 일도 하지 않으면서 소모

왜 HW의 도움이 필요한가?

- 다른 시도 SW적 시도들
  - 피터슨 알고리즘
    - 2 개의 쓰레드에서만 동작 (id가 0과 1)

```
volatile int victim = 0;
volatile bool flag[2] = {false, false};

Lock(int myID)
{
        int other = 1 - myID;
        flag[myID] = true;
        victim = myID;
        while(flag[other] && victim == myID) {}
}
Unlock (int myID)
{
        flag[myID] = false;
}
```

왜 HW의 도움이 필요한가?

- 다른 시도 SW적 시도들
  - 빵집 알고리즘
    - n개의 쓰레드에서만 동작 (id가 0과 1)

```
int choosing[n], turn[n];
프로세스 i의 코드: 0 <= i < n:

lock(int i) {
   choosing[i] = 1;
   turn[i] = max(turn[0], turn[1], ..., turn[n-1]) + 1;
   choosing[i] = 0;
   for (int j = 0; j < n; j++)
     if (j != i) {
       while (0 != choosing[j]) ;
       while (turn[j] != 0 && (turn[j], j) < (turn[i], i)) ;
     }
}
unlock(int i) {
   turn[i] = 0;
}

// (a, b) < (c, d) 는 (a < c) || (a = c && b < d) 을 뜻함
```

왜 HW의 도움이 필요한가?

- SW의 문제
  - 제한된 프로세스의 개수
  - 너무 복잡한 연산으로 인한 delay
  - 현대 CPU에서는 제대로 동작하지 않음
    - 추가적인 CPU제어 필요
- 따라서 Hardware의 도움이 필요하다.
  - *test-and-set* 명령어, 다른 말로 원자적 교환(*atomic exchange)*

Test And Set (Atomic Exchange)

- 원자적 교체 (Atomic Exchange)
- 락을 간단하게 구현하기 위한 명령어

```
1    int AtomicExchange(int *ptr, int new) {
2        int old = *ptr;   // fetch old value at ptr
3        *ptr = new;       // store 'new' into ptr
4        return old;       // return the old value
5    }
```

  - ptr가 가리키는 이전 값을 리턴(TEST)한다.
  - ***동시에*** 그 값을 new로 바꾼다(SET).
  - 이동작은 원자적으로 수행된다.
  - SW만으로는 구현할 수 없다.
  - CPU가 메모리 버스를 잠가서 다른 CPU나 코어의 메모리 관련 명령어의 실행을 원천봉쇄한다.

test-and-set을 사용한 스핀 락

- 스핀 락(spin lock)

```
class mutex {
   bool flag;
public:
   mutex() {
      flag = false;
      // false는 락이 획득가능함을, true는 누가 락을 획득했음을 나타냄
   }
   void lock()
   {
      while (true == AtomicExchange(&flag, true))
      ;  // 스핀 (아무 일도 하지 않음)
   }
   void unlock()
   {
      flag = false; // 락을 반환
   }
}
```

스핀 락 평가

- **정확성**: 그렇다
  - 오직 한 쓰레드 만이 임계영역에 진입할 수 있다.
- **공정성**: 아니다
  - 스핀 락은 어떠한 공정성도 제공하지 않는다.
  - 사실상 영원히 스핀하는 쓰레드가 생길 수 있다.
- **성능**:
  - 단일 CPU에서의 성능은 끔찍할 수 있다.
    - Context Switch가 발생할 때까지 계속 SPIN
  - 쓰레드의 개수가 CPU의 개수보다 작거나 같을 경우, 스핀락의 성능은 적절하다.

Compare-And-Swap

- 주소(ptr)에 있는 값이 기대 값(expected)과 같은지 본다.
  - *같다면*, 주소(ptr)의 값을 새 값(new)으로 수정한다.
  - 주소(ptr)의 원래 메모리 값을 리턴한다.

```
1    int CompareAndSwap(int *ptr, int expected, int new) {
2        int actual = *ptr;
3        if (actual == expected)
4                *ptr = new;
5        return actual;
6    }
```

Compare-and-Swap hardware atomic instruction (C-style)

```
1    void mutex::lock() {
2        while (CompareAndSwap(&flag, false, true) == true)
3                ; // 스핀
4    }
```

Spin lock with compare-and-swap

Compare-And-Swap (계속)

- x86 기계어 구현

```
int CompareAndSwap(volatile int *addr, int old_v, int new_v)
{
    int ret_v;
    int temp = reinterpret_cast<int>(add)
    _asm    mov esi, temp;
    _asm    mov eax, old_v;
    _asm    mov ebx, new_v;
    _asm    lock cmpxchg [esi], ebx;
    _asm    mov ret_v, eax;
    return ret_v;
}
```

Compare-And-Swap (계속)

- C++11 구현

```
bool CompareAndSwap(atomic_bool *addr,
                    bool expected, bool new_val)
{
   atomic_compare_exchange_strong(addr, &expected, new_val);
   return expected;
}
```

  - atomic_bool : bool 클래스인데 원자적이다.
    - 읽고 쓰는 것이 원자적으로 수행되는 변수
    - bool과 똑같이 사용하며 자세한 차이점은 4학년 때
  - atomic_compare_exchange_strong
    - 성공하면 true, 실패하면 false를 리턴
    - expected에 addr에 있던 원래 값을 넣어서 리턴

Compare-And-Swap (계속)

```
#include <atomic>
#include <thread>
#include <iostream>
using namespace std;

static const int NUM_THREADS = 4;
volatile int sum;
atomic_bool lock = false;

bool CompareAndSwap(atomic_bool *addr, bool expected, bool new_val)
{
   atomic_compare_exchange_strong(addr, &expected, new_val);
   return expected;
}

void my_lock()
{
   while (true == CompareAndSwap(&lock, false, true));
}

void my_unlock()
{
   lock = false;
}
```

Compare-And-Swap (계속)

```
void worker(int count)
{
   for (int i = 0; i < count; ++i) {
      my_lock();
      sum = sum + 2;
      my_unlock();
   }
}

int main()
{
   thread workers[NUM_THREADS];
   sum = 0;
   for (auto &th : workers) th = thread{ worker, 50000000 / NUM_THREADS };
   for (auto &th : workers) th.join();
   cout << "Sum = " << sum << endl;
   system("pause");
}
```

Load-Linked와 Store-Conditional

```
1    int LoadLinked(int *ptr) {
2        return *ptr;
3    }
4
5    int StoreConditional(int *ptr, int value) {
6        if (*ptr에 LoadLinked를 실행한 이후로 아무도 *ptr에 쓰지 않았으면) {
7            *ptr = value;
8            return 1;        // success!
9        }
10       else return 0;       // failed to update
11   }
```

Load-linked And Store-conditional

- Load-Linked로 주소의 값을 읽었을 경우 Store-conditional은 아무도 그 주소에 쓰지 않았을 경우에만 저장한다.
  - **성공**: ptr주소의 값을 value로 수정하고 1을 리턴
  - **실패**: ptr주소의 값은 수정되지 않고 0을 리턴

Load-Linked와 Store-Conditional

```
1    void lock(lock_t *lock) {
2        while (1) {
3                while (LoadLinked(&lock->flag) == 1)
4                        ; // spin until it's zero
5                if (StoreConditional(&lock->flag, 1) == 1)
6                        return; // if set-it-to-1 was a success: all done
7                                otherwise: try it all over again
8        }
9    }
10
11   void unlock(lock_t *lock) {
12       lock->flag = 0;
13   }
```

Using LL/SC To Build A Lock

```
1    void lock(lock_t *lock) {
2        while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
3                ; // spin
4    }
```

A more concise form of the `lock()` using LL/SC

Load-Linked와 Store-Conditional

- LL-SC를 제공하는 CPU
  - Alpha: ldl_l/stl_c and ldq_l/stq_c
  - PowerPC: lwarx/stwcx and ldarx/stdcx
  - MIPS: ll/sc
  - ARM: ldrex/strex (ARMv6 and v7), and ldxr/stxr (ARM version 8)
  - 인텔 계열 CPU는 제공하지 않는다.
- LL-SC의 장단점
  - 장점 : Compare&Swap보다 강력하다.
    - 4학년(멀티코어 프로그래밍) 에서 배움
  - 단점 : 코드가 복잡해진다. 성능이 떨어진다. 성공해야 하는데 실패할 수 있다… **하지만 C&S와 별 차이 없다.**
    - 성공해야 하는데 실패하는 경우도 있어서 ace에 strong, weak가 존재한다.

Fetch-And-Add

- 원래 값을 리턴하면서 값을 증가시키는 동작을 원자적으로 수행.

```
1    int FetchAndAdd(int *ptr) {
2        int old = *ptr;
3        *ptr = old + 1;
4        return old;
5    }
```

Fetch-And-Add Hardware atomic instruction (C-style)

Fetch-And-Add

- C++11에서 구현

```
1   #include <atomic>
2
3   int FetchAndAdd(std::atomic_int &ptr)
4   {
5      return ptr++;
6   }
```

Fetch-And-Add implementation (C++11)

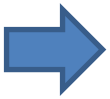
```
1   #include <atomic>
2
3   int FetchAndAdd(std::atomic_int &ptr, int num)
4   {
5      return ptr.fetch_add(num);
6   }
```

Fetch-And-Increment가 아닌 Fetch-And-Add implementation (C++11)

Fetch-And-Add

- C++11의 Atomic 자료구조
  - 기본 자료 구조의 일부를 원자적으로 구현
    - `char, int, short, bool`
  - `#include <atomic>`으로 사용
  - 읽기, 쓰기, +=, -=, ++, -- 연산을 원자적으로 수행

```
volatile int sum;
void worker(int count)
{
   for (int i = 0; i < count; ++i) {
      my_lock();
      sum = sum + 2;
      my_unlock();
   }
}
```



```
atomic_int sum;
void worker(int count)
{
   for (int i = 0; i < count; ++i)
      sum += 2;
}
```

티켓 락

- **티켓 락(Ticket lock)**은 fetch-and add로 구현할 수 있다.
  - 락을 신청한 순서대로 락을 얻는다. → 공정성

```
1   class mutex {
2         atomic_int ticket;
3         atomic_int turn;
4
5   public:
6   mutex() {
7         ticket = 0;
8         turn = 0;
9    }
10
11  void lock() {
12        int myturn = ticket++;
13        while (turn != myturn)
14                ; // spin
15   }
16  void unlock() {
17        turn++;
18   }
19  };
```

과도한 스핀

- HW기반의 스핀 락은 간단하고 확실히 동작한다.

- 하지만 많은 경우, 이러한 방법은 비효율적이다.
  - 락을 가진 쓰레드가 준비상태가 되면, 기다리는 쓰레드들은 **전체 타임슬라이스**를 락의 상태를 검사하는데 **낭비**할 수 있다.
  - 호위현상 (Convoying)이라고도 한다.
    - Lock을 가진 쓰레드가 스케줄링이 되지 않았을 때 다른 쓰레드들이 계속 기다리는 현상

How To Avoid *Spinning*?
We'll need OS Support too!

간단한 해결: 양보

- 스핀하게 되면, CPU를 다른 쓰레드에 양보한다.
  - OS 호출을 통해 *실행* 상태에서 *준비* 상태로 변환한다.
  - **문맥 교환** 비용이 상당하며, **기아(starvation)**문제가 계속 존재한다.

```
1    clasee mutex {
2    atomic_bool flag;
3    public:
4    mutex() {    // 생성자
5        flag = false;
6    }
7
8    void lock() {
9       while (!atomic_compare_exchange_strong(&flag, false, true))
10           std::this_thread::yield(); // give up the CPU
11   }
12
13   void unlock() {
14      flag = false;
15   }
16   };
```

Lock with Test-and-set and Yield

큐의 사용: 스핀대신 잠자기

- Yield()의 아쉬움
  - Unlock이 되지 않았는데도, 깨어나서 다시 yield()를 실행할 경우가 많음.
- 해결 방법
  - 프로그래밍으로 해결하자.
  - OS의 도움이 필요하다.
    - yield()대신 대기상태로 하는 API : park()
    - 대기상태의 쓰레드를 깨우는 API : unpark()
  - 대기상태의 쓰레드가 여러 개면?
    - park()할때 queue에 thread_id를 넣자.
    - unpark()를 unpark(thread_id)로 하자.

큐의 사용 : 스핀대신 잠자기

```
1    class mutex {
2       atomic_bool flag, guard;
3       queue <std::thread::id> q;
4    public:
5       mutex() : flag (false), guard(false) {}
6       void lock() {
7           while (atomic_compare_exchange_strong(&guard, false, true))
8               ; // acquire guard lock by spinning
9           if (flag == false) {
10              flag = true; // lock is acquired
11              guard = false;
12          } else {
13              q.push(std::this_thread::get_id());
14              guard = false;
15              park();
16          }
17   }
18   …
```

Lock With Queues, Test-and-set, Yield, And Wakeup

큐의 사용 : 스핀대신 잠자기

```
22   void unlock() {
23       while (!atomic_compare_exchange_strong(&guard, false, true))
24           ; // acquire guard lock by spinning
25       if (q.empty())
26           flag = false; // let go of lock; no one wants it
27       else
28           unpark(q.top()); // hold lock (for next thread!)
29           q.pop();
30       guard = false;
31   }
32 }
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

Park/Unpark 문제

- 무한 대기
  - (*thread B*)가 `park()`를 호출하기 바로 전에 (*thread A*)가 락을 반납하면? → 쓰레드 B는 영원히 대기상태.
- `setpark()`를 사용해 해결
  - 호출하는 쓰레드는 곧 `park`할 것임을 미리 알려준다.
  - 공교롭게도 인터럽트가 발생해서 다른 쓰레드가 그 사이에 `unpark`를 호출했다면, `park`호출은 쓰레드를 대기시키지 않고 즉시 복귀한다.

```
1            q.push(std::this_thread::get_id());
2            setpark(); // new code
3            guard = false;
4            park();
```

Code modification inside of `lock()`

- **이러한 문제점으로 인해 C++11의 표준이 아님**
  - SOLARIS운영체제에서만 존재

Futex

- 리눅스는 futex를 제공한다.(Solaris의 `park, unpark`와 비슷).
  - `futex_wait(address, expected)`
    - 호출하는 쓰레드를 대기상태로 전환
    - 만약 `address`에 저장되어 있는 값이 `expected`와 다르다면 즉시 리턴.
  - `futex_wake(address)`
    - 큐에서 대기하고 있는 쓰레드 중 하나를 깨운다. (준비 상태로 바꾼다.)
- 역시 C++11 표준이 아님

C++11 해결 방식

- 대기와 재시작을 포함한 효율적인 mutex의 구현은 사용자에게 맡기면 안되고 mutex가 기본적으로 가져야 할 특성이다.
- C++11에서는 다음 페이지에 있는 2단계 락을 포함해 구현되었다.
- 하지만 대기와 재시작의 개념은 다른 용도로 사용할 수 있기 때문에 **조건변수**에서 다룬다.

2단계 락

- 락이 *금방* 해제된다면 스피닝이 더 좋다는 현실을 구현한 것이 2단계 락(two-phase locking)
  - **첫 단계**
    - 락을 금방 얻을 수 있을 때를 위해 잠시 동안 스핀한다.
    - 스핀으로 락을 얻지 못할 경우 두번째 단계(second phase)로 이행.
  - **두번째 단계**
    - 호출하는 쓰레드를 대기상태로 변환한다.
    - 락이 해제되었을 경우 대기상태에서 깨운다.

29. 락 기반의 병행 자료 구조

락 기반 병행 자료 구조

- 자료 구조에 락을 추가하면 쓰레드 안전(**thread safe)** 자료구조가 된다.
  - 락을 어떻게 추가했느냐 에 따라, 그 자료구조의 정확성과 성능 이 결정된다.

예 : 락이 없는 병행 카운터

- 간단하지만 멀티쓰레드에서 사용 불가능

```
1    class counter_t {
2       int value;
3    public:
4       counter_t() { value = 0; }
8
9       void increment() { value++; }
12
13      void decrement() { value--; }
16
17      int get() { return value; }
19   };
```

예 : 락을 사용한 병행 카운터

- **락** 추가
  - 락을 획득한 이후에만 내부 자료구조를 변경한다.

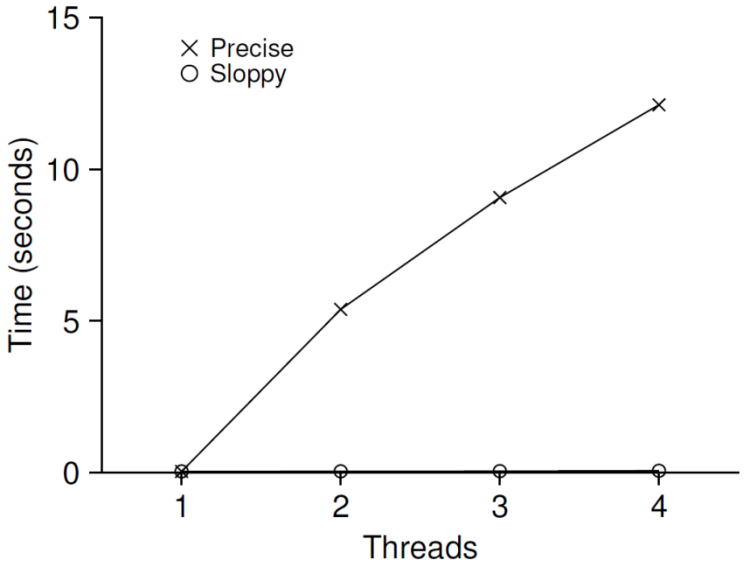
```
1    class counter_t {
2       int value;
3       mutex c_lock;
4    public:
5       counter_t() { value = 0; }
8
9       void increment() {
10         c_lock.lock();
11         value++;
12         c_lock.unlock();
11      }
```

예 : 락을 사용한 병행 카운터

```
13      void decrement() {
14          c_lock.lock();
15          value--;
14          c_lock.unlock();
15       }
16
17      int get() {
18          int temp;
19          c_lock.lock();          // 락을 걸 필요가 꼭 있을까?
20          temp = value;           //     필요하다. (compiler, cpu 문제)
21          c_lock.unlock();        //     하지만 대부분의 프로그램에서는 문제 없다.
18          return temp;
19      }
20  };
```

락의 문제점 : 성능

- 하나의 카운터를 여러개의 쓰레드에서 증가 시킨다.
  - 쓰레드마다 백만번씩.
  - 쿼드 코어 인텔 2.7GHz i5 CPU의 iMac



Performance of
락 카운터 vs. 엉성한 카운터
(Threshold of Sloppy, S, is set to 1024)

Synchronized counter scales poorly.

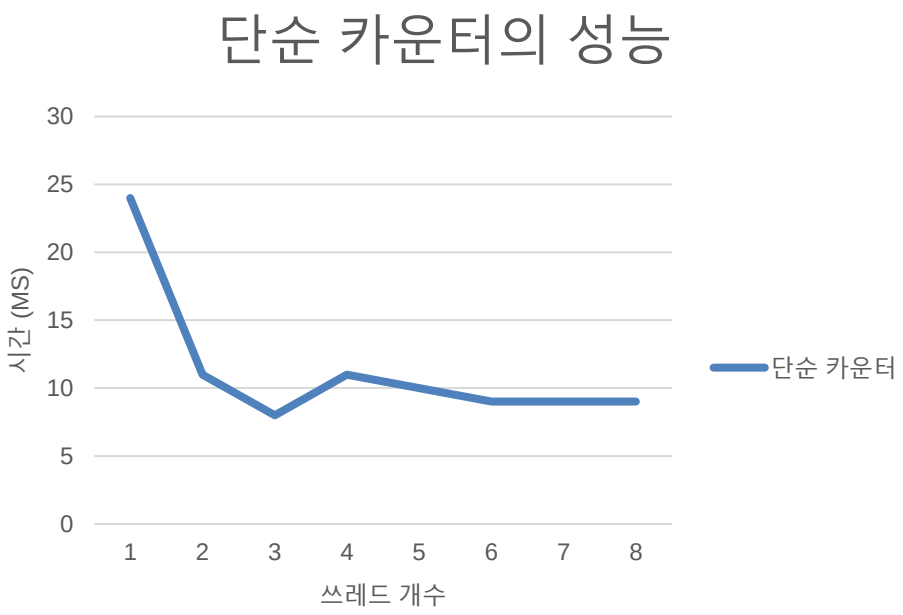
실제 성능 비교

- 벤치 마크 프로그램

```
#include <thread>
#include <iostream>
#include <chrono>
using namespace std;
using namespace chrono;

class counter {…};
counter c;

void thread_func(int loops)
{
   for (int i = 0; i < loops; ++i) c.increment();
}

int main()
{
   thread th[8];
   for (int num_th = 1; num_th <= 8; ++num_th) {
      c.reset();
      auto st_t = high_resolution_clock::now();
      for (int j = 0; j < num_th; ++j)
         th[j] = thread{ thread_func, 10000000 / num_th };
      for (int j = 0; j < num_th; ++j) th[j].join();
      auto end_t = high_resolution_clock::now();
      auto t = end_t - st_t;
      cout << "Time = " << duration_cast<milliseconds>(t).count();
      cout << "   Result = " << c.get() << endl;
   }
}
```

성능 비교

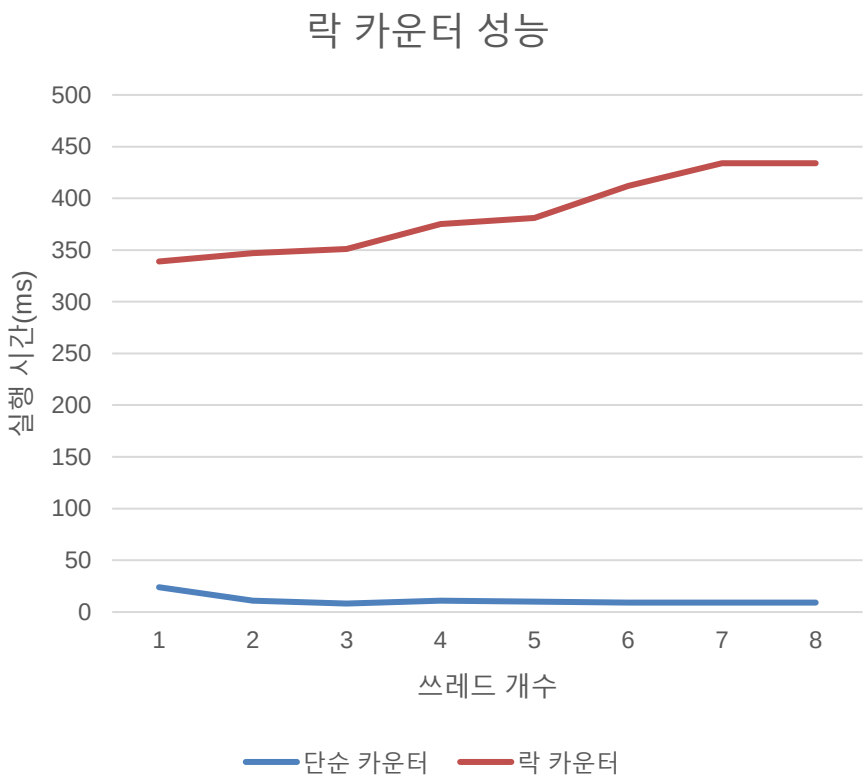
- 락을 사용하지 않았을 경우의 성능
  - i7 4702HQ (quad core)
  - **수행 결과 부정확**




```
class counter
{
    volatile int value;
    mutex c_lock;
public:
    counter() { value = 0; }
    void increment() { value++; }
    void decrement() { value--; }
    int get() { return value; }
    void reset() { value = 0; }
};
```

성능 비교

- 락을 사용했을 경우의 성능
  - i7 4702HQ (quad core)
  - 정확한 결과값이 나오기는 함




```
class counter
{
    volatile int value;
    mutex c_lock;
public:
    counter() { value = 0; }
    void increment() {
        c_lock.lock();
        value++;
        c_lock.unlock();
    }
    void decrement() {
        c_lock.lock();
        value--;
        c_lock.unlock();
    }
    int get() { return value; }
    void reset() { value = 0; }
};
```

락 카운터

- 모니터(monitor)와 유사한 구현
  - 모니터는 모든 메소드들의 실행을 자동적으로 상호배제 하는 자료구조
- 단점 : 성능이 너무 떨어진다.
  - 더 느려진다!!!
  - 적어도 느려지면 안되고, 쓰레드와 CPU개수가 늘어날수록 성능이 향상되어야 한다.
  - 완벽한 확장성(Perfect Scaling)이라는 단어는 잘 안쓰는 단어이다.

엉성한 카운터

- 락카운터의 성능 개선
  - 락의 사용 회수를 줄이자!!
    - 전역 변수의 사용 회수를 줄이자
    - 대신 쓰레드 지역변수를 사용하자
- 쓰레드 지역 변수 (thread local storage, TLS)
  - 같은 전역 변수인데 쓰레드 마다 다른 장소에 접근하는 변수
  - 초 간단 구현
    - 배열로 구현하고 쓰레드 마다 자신의 ID로 indexing 한다.
  - C++11의 구현
    - `thread_local` 확장자 사용

엉성한 카운터

- 엉성한 카운터는…
  - 쓰레드당 하나의 **지역 카운터**가 존재
  - 하나의 **전역 카운터**가 존재.
  - 여러 개의 **락**:
    - 지역 카운터당 한 개, 전역 카운터를 위한 한 개

- 예: 4개의 CPU(또는 core)를 갖는 컴퓨터
  - 같은 개수의 쓰레드를 실행
  - 4개의 지역 카운터
  - 1개의 전역 카운터

엉성한 카운터의 구현

- 각 쓰레드는
  - 지역 카운터를 하나씩 갖는다.
  - 각 CPU(또는 core)는 자신의 지역 카운터를 갖는다.
    - 쓰레드의 개수와 CPU의 개수가 같아서 각 CPU가 한 개씩의 쓰레드를 실행하므로.
    - 자신의 지역 카운터의 수정은 아무런 충돌이 없음
    - 따라서 카운터 수정은 확장성이 있다.
  - 지역 카운터 값은 주기적으로 전역 카운터에 더해진다.
    - 전역 락을 얻는다.
    - 지역 카운터의 값만큼 증가시킨다.
    - 이때 지역 카운터의 값은 다시 0으로 초기화 한다.

엉성한 카운터의 구현

- 얼마나 자주 지역에서 전역으로의 전송이 필요한 가는 한계치 s (엉성수치)로 결정
  - s가 작아질수록:
    - 일반 락 카운터와 비슷해진다.
  - s가 커질수록:
    - 확장성이 커진다.
    - 전역 카운터 값과 실제 값과의 차이가 커진다.

엉성한 카운터 예

- 엉성한 카운터의 변화 추적
  - 한계치 S는 5.
  - 4개 쓰레드가 4개 코어에서 실행
  - 각 쓰레드는 자신의 지역 카운터 $L_1 \ldots L_4$를 수정

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from)
7	0	2	4	5 → 0	10 (from)

엉성한 카운터구현 (OLD)

```
1       class counter_t {
2           int global;             // global count
3           std::mutex glock;       // global lock
4           int local[NUMCPUS];     // local count (per cpu)
5           std::mutex llock[NUMCPUS]; // ... and locks
6           int threshold;          // update frequency
7       public:
8
9       // 생성자: record threshold, init locks, init values
10      //       of all local counts and global count
11      counter_t(int threshold_v) {
12          threshold = threshold_v;
13          global = 0;
18          for (int i = 0; i < NUMCPUS; i++) local[i] = 0;
22      }
23
```

엉성한 카운터 구현 (OLD)

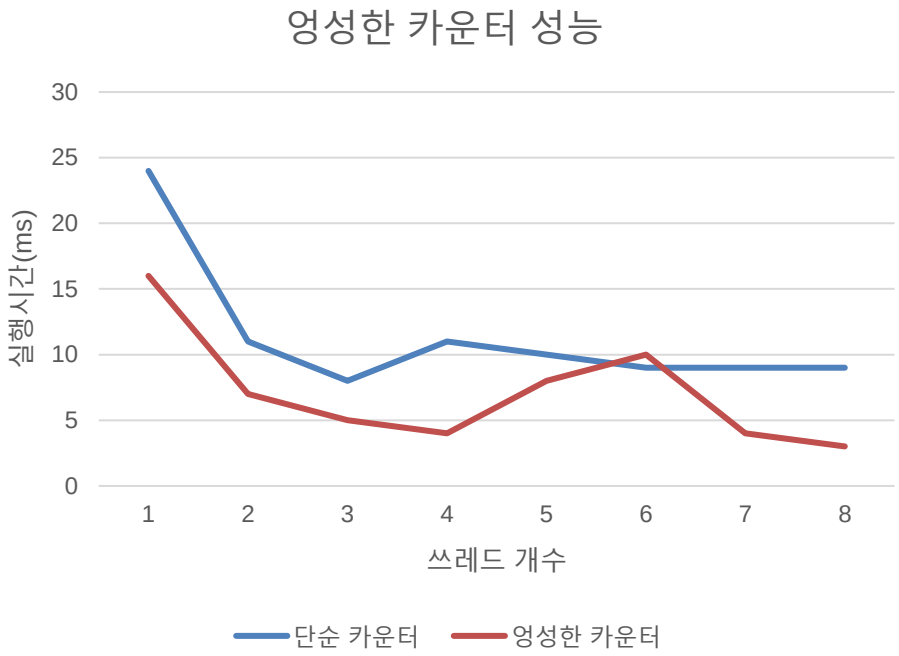
```
(Cont.)
24      // update: usually, just grab local lock and update local amount
25      //         once local count has risen by 'threshold', grab global
26      //         lock and transfer local values to it
27      void update(int threadID, int amt) {
28          llock[threadID].lock();
29          local[threadID] += amt;          // assumes amt > 0
30          if (local[threadID] >= threshold) { // transfer to global
31              glock.lock();
32              global += local[threadID];
33              glock.unlock();
34              local[threadID] = 0;
35          }
36          llock[threadID].unlock();
37      }
38
39      // get: just return global amount (which may not be perfect)
40      int get() {
41          int temp;
41          glock.lock();
42          temp = global;
43          glock.unlock();
44          return temp;   // only approximate!
45      }
```

Sloppy Counter 구현 (C++11)

```
static const int NUMCPUS = 8;

thread_local int local; // local count (per cpu)
class s_counter {
   atomic_int global; // global count
   int threshold; // update frequency
public:
   s_counter(int th) { threshold = th; global = 0; }
   void increment() {
      local++;
      if (local >= threshold) { global += local; local = 0; }
   }
   int get() { return global; }
   void flush() { global += local; local = 0; }
   void reset() { global = 0; }
};

s_counter c{ 10000 };

void thread_func(int loops)
{
   for (int i = 0; i < loops; ++i) c.increment();
   c.flush();
}
```

Sloppy Counter 구현 (C++11)

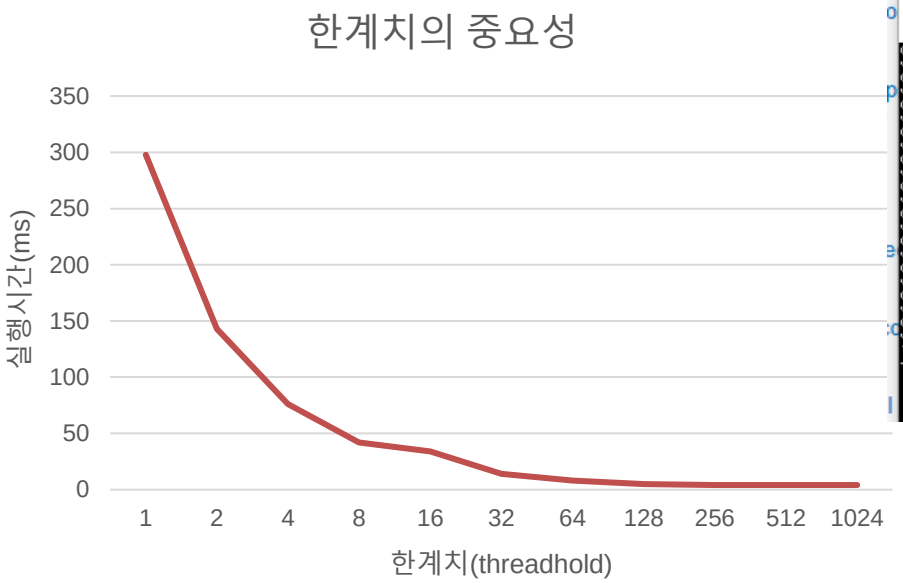
- 성능

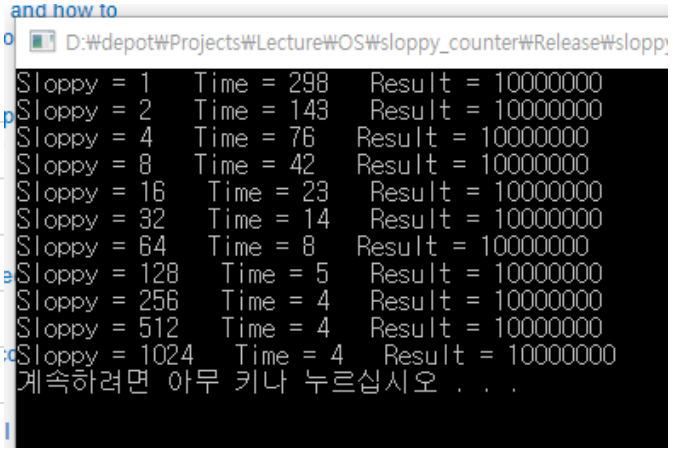


  - 성능이 왜 더 좋을까?
  - std::thread_local과 local[NUM_CPU]와의 차이?

한계치 s의 중요성

- 4개의 쓰레드가 각각 카운터를 백만번 증가시킨다.
  - 작은 S → **낮은** 성능, 전역 카운터는 비교적 **정확**
  - 큰 S → **훌륭한** 성능, **뒤처진** 전역 카운터





Scaling Sloppy Counters

병렬 연결 리스트

- 포인터로 연결된 리스트
- 삽입 연산만 구현
- new를 사용한 메모리 할당 시 오류 처리
  - 락의 해제를 빼놓지 않도록 신경 써야 한다.

병행 연결 리스트

```
1 class NODE {              // basic node structure
2  public:
3    int key;
4    NODE *next;
5  };
6
8   class LIST {            // basic list structure (one used per list)
9   public:
9      NODE *head;
10     std::mutex lock;
11
13  LIST() {
14     head = nullptr;
16  }
17
18  bool Insert(int key) {
19                lock.lock();
20                NODE *p = new NODE;
21                if (p == nullptr) {
22                        cout << "Can't allocate memory.");
23                        lock.unlock();
24                        return false; // fail
25       }
```

병행 연결 리스트

```
(Cont.)
26                  p->key = key;
27                  p->next = head;
28                  head = p;
29                  lock.unlock();
30                  return true; // success
31          }
32
33          bool List_Lookup(int key) {
34                  lock.lock();
35                  NODE *curr = head;
36                  while (nullptr != curr) {
37                          if (curr->key == key) {
38                                  lock.unlock();
39                                  return true; // success
40                          }
41                          curr = curr->next;
42                  }
43                  lock.unlock();
44                  return false; // failure
45          }
```

병렬 연결 리스트(Cont.)

- 삽입 함수 시작 시 락을 **획득**한다.
- 복귀하기 바로 전에 그 락을 **반납**한다.
  - `new`가 실패할 경우에도 삽입 함수에서 복귀하기 전에 락을 반납해야 한다.
  - 임계 영역의 크기가 필요이상으로 커서 병행성이 떨어진다.
  - 이러한 예외 처리는 오류를 포함하기 쉽다.
  - **해결책**: 락의 획득과 반납은 실제 임계영역 *만*을 둘러싸야 한다.

병행 연결 리스트: 재작성

```
18  bool Insert(int key) {
19  // synchronization not needed
20     NODE *p = new NODE;
21     if (p == nullptr) {
22        cout << "Can't allocate memory.");
23        return false;
24     }
25     p->key = key;
26     // just lock critical section
27     lock.lock();
28     p->next = head;
29     head = p;
30     lock.unlock();
31     return true;
32  }
```

병행 링크드 리스트: 재작성

```
(Cont.)
22        bool List_Lookup(int key) {
23                bool rv = false;
24                lock.lock();
25                NODE *curr = head;
26                while (nullptr != curr) {
27                        if (curr->key == key) {
28                                rv = true;
29                                break;
30                        }
31                        curr = curr->next;
32                }
33                lock.unlock();
34                return rv; // now both success and failure
35        }
```

연결 리스트의 병행성 개선

- Hand-over-hand locking (lock coupling)
  - 전체 리스트를 하나의 락으로 관리하지 않고, **노드 마다 락**을 갖는다.
  - 리스트를 순회할 때,
    - 다음 노드의 락을 획득하고.
    - 현재 노드의 락을 반납한다.
  - 높은 병행성으로 리스트를 관리할 수 있다.
    - 하지만, 실제로는, 락 획득과 반납 오버헤드 때문에 기본 성능이 매우 낮다.

Michael과 Scott의 병행 큐

- 두개의 락이 있다.
  - 하나는 **head**, 하나는 **tail**용.
  - 두 개의 락으로 *enqueue*와 *dequeue*의 병렬성을 얻는다.
- 더미 노드의 추가(또는 보초노드)
  - 큐가 초기화 될 때 할당.
  - head와 tail연산을 서로 분리한다.

Concurrent Queues (Cont.)

```
1         struct NODE {
2                 int value;
3                 NODE *next;
4         };
5
6         class QUEUE {
7         public:
8                 NODE *head;
8                 NODE *tail;
9                 std::mutex headlock, tailLock;
11
13        QUEUE() {
16                q->head = q->tail = new NODE;
19        }
20
21        void Queue_Enqueue(int value) {
22                NODE *tmp = new NODE;
23                assert(tmp != nullptr);
24                tmp->value = value;
```

Concurrent Queues (Cont.)

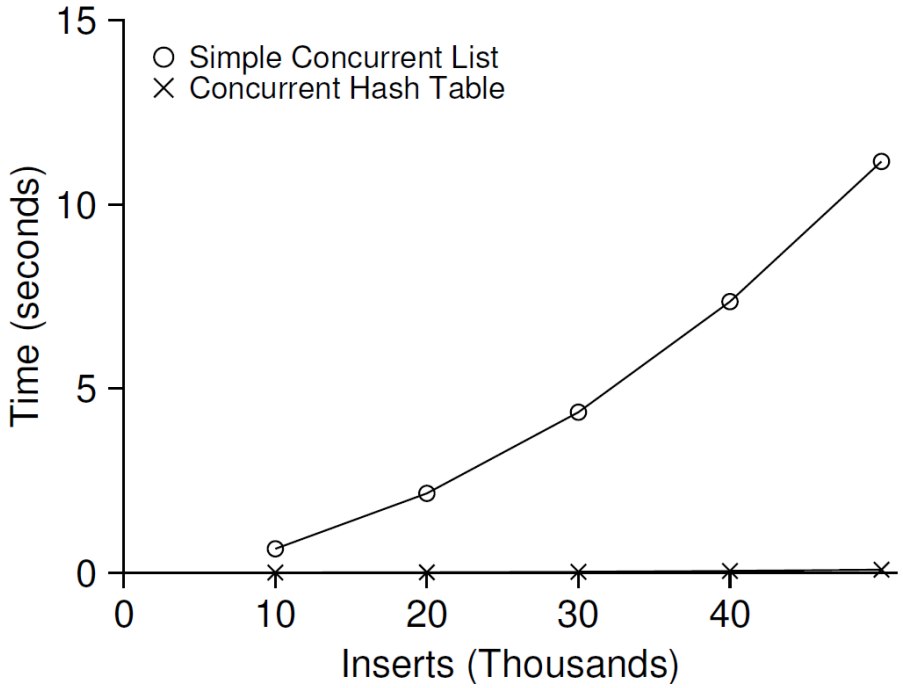
```
(Cont.)
25                tmp->next = nullptr;
26
27                tailLock.lock();
28                tail->next = tmp;
29                tail = tmp;
30                tailLock.unlock();
31        }
32
33        int Queue_Dequeue(int *value) {
34                headlock.lock();
35                NODE *tmp = head;
36                NODE *newHead = tmp->next;
37                if (newHead == nullptr) {
38                        headlock.unlock();
39                        return -1; // queue was empty
40                }
41                *value = newHead->value;
42                head = newHead;
43                headlock.unlock();
44                delete tmp;
45                return 0;
46        }
```

병행 해쉬 테이블

- 간단한 해쉬 테이블
  - 크기 고정(size가 변경되지 않음)
  - 병행 연결 리스트로 구현
  - 해시 버켓마다 락이 있다.
    - 전역 락이 없다.

병행 해시 테이블의 성능

- From 10,000 to 50,000 concurrent updates from each of four threads.
  - iMac with four Intel 2.7GHz i5 CPUs.



The simple concurrent hash table scales magnificently.

Concurrent Hash Table

```
1       #define BUCKETS (101)
2
3       class HASH_TABLE {
4               LIST lists[BUCKETS];
5       public:
13
14      int Hash_Insert(int key) {
15              int bucket = key % BUCKETS;
16              return lists[bucket].List_Insert(key);
17      }
18
19      int Hash_Lookup(int key) {
20              int bucket = key % BUCKETS;
21              return lists[bucket].List_Lookup(key);
22      }
```

병행 자료 구조

- 락 획득과 해제 시 코드의 흐름에 주의를 기울여야 한다.
  - `lock_guard`를 사용해야 하는 이유.
- 임계영역의 크기를 최소화 해야 한다.
  - `lock_guard`사용시 주의
- 병행성 개선이 반드시 성능 개선으로 이어지지 않는다.
  - 추가 오버헤드에 주의
- 미숙한 최적화(premature optimization)주의
  - 전체 성능에 미치는 영향을 계속 살펴야한다.
- 논 블럭킹(non—blocking)자료 구조
  - 성능을 위한 최선의 선택
  - 너무 어렵다, 4학년 멀티코어 프로그래밍에서 다룸.

숙제 7

- **다음 병행 알고리즘들을 구현하고 성능을 비교하라.**
  - 병행 연결리스트
    - 그림 29.7의 연결 리스트, 그림 29.8의 개선 연결 리스트, hand-over-hand locking 연결 리스트의 3개를 구현하라.
    - 쓰레드의 개수와 그 성능을 <그림 29.3>과 같이 그리도록 하라.
      - 쓰레드 개수는 1에서 10까지, 100만개의 랜덤 값을 삽입 (0에서 10000까지의 랜덤한 정수)
      - 삽입 하는 값을 전역 변수 SUM에 더하고, 실행 후 리스트의 값을 전부 더해서 SUM과 비교하라.
  - 병행 해시 테이블
    - 병행 해시 테이블 A와 B를 구현하여, 성능 비교를 하여라
      - A는 mutex하나로 전체 insert와 lookup을 상호 배제하는 구현
        - » 버켓을 이루는 리스트에 Lock을 넣지 않는다.
      - B는 위에서 구현한 29.8 개선 연결 리스트를 버킷으로 사용한 해시 테이블
        - » 전체 버킷을 상호배제하는 mutex는 없다.
    - 쓰레드 개수는 1에서 10까지, 100만개의 랜덤 값을 삽입 (0에서 10000까지의 랜덤한 정수)
      - 버킷의 개수는 1000개이고 Hash 함수는 / 10이다.
    - 삽입 하는 값을 전역 변수 SUM에 더하고, 실행 후 리스트의 값을 전부 더해서 SUM과 비교하라.

숙제 7

- **다음 병행 알고리즘들을 구현하고 성능을 비교하라.**
  - 벤치마크

```
For (int = i; i < 1000000 / num_threads ; ++i)
{
    if (rand() % 2) {
        hash_table.insert(rand() % 10000);
    } else
        hash_table.lookup(rand() % 10000);
   }
}
```