분산 시스템 HW1-2:
CPU Synchronization Operation

이현우 (2015-21259)
인터넷 융합 및 보안 연구실
(hwlee2014@mmlab.snu.ac.kr)

March 20, 2017

1 CPU 정보 및 내용

1.1 CPU 사양

필자가 사용한 머신의 CPU 사양은 아래와 같다.

- **모델명** Intel(R) Xeon(R) CPU E3-1245 v3 @ 3.40GHz
- **물리 CPU 개수** 1개 (processor id의 종류가 1개)
- **논리 CPU 개수** 8개 (processor 번호가 0-7)
- **Core 개수** 4개 (core id 번호가 0-4)
- **Siblings** 8개 (hyperthreading 사용)

1.2 CPU 지원 내용

[1]에는 x86_64 아키텍처에서 지원하는 명령어들이 나열되어 있다. 이 중 Synchronization과 관련된 Primitive들이 여러 가지 있는데 이 중 LOCK을 Prefix로 가질 수 있는 것은 다음과 같다.

- **ADC** Add with Carry를 뜻하며, 목적지 피연산자와 소스 피연산자, 그리고 캐리 플래그를 더하여 목적지 주소에 저장한다. LOCK과 함께 쓰임으로써, 메모리로부터 읽어와서 덧셈을 취하고, 저장하는 것까지 원자적으로 수행된다.
- **ADD** 목적지 피연산자와 소스 피연산자를 더하고 목적지 주소에 저장한다.
- **AND** 목적지 피연산자와 소스 피연산자에 비트마다의 논리적 AND를 수행한 뒤, 목적지 주소에 저장한다.
- **BTC** Bit Test and Complement를 뜻하며, 첫번째 피연산자인 비트 스트링에서 두번째 피연산자인 오프셋을 활용하여 비트를 선택하고 이 값을 CF 플래그에 저장한 뒤, 이의 보수를 취한다.
- **BTR** Bit Test and Reset을 뜻하며, 선택된 비트를 0으로 설정한다.
- **BTS** Bit Test and Set를 뜻하며, 선택된 비트를 1로 설정한다.
- **CMPXCHG** Compare and Exchange를 뜻하며, AL, AX, EAX, 혹은 RAX 레지스터와 첫번째 피연산자(목적지 피연산자)를 비교하며, 값이 동일하면, 두번째 피연산자(소스 피연산자)가 목적지 피연산자로 로드된다.
- **CMPXCHG8B** CMPXCHG와 동일한 연산을 수행하되 8바이트에 대해서 수행한다.
- **CMPXCHG16B** CMPXCHG와 동일한 연산을 수행하되 16바이트에 대해서 수행한다.

- **DEC** 목적지 피연산자의 값을 1 감소시켜서 목적지 주소에 저장한다.
- **INC** 목적지 피연산자의 값을 1 증가시켜서 목적지 주소에 저장한다.
- **NEG** 목적지 피연산자의 2의 보수를 취하고 목적지 주소에 저장한다.
- **NOT** 목적지 피연산자의 1의 보수를 취하고 목적지 주소에 저장한다.
- **OR** 목적지 피연산자와 소스 피연산자 사이에 논리적 OR 연산을 수행하고 목적지 주소에 저장한다.
- **SBB** Integer Substraction with Borrow를 뜻하며, 소스 피연산자와 캐리 플래그를 더한 뒤, 이를 목적지 피연산에서 뺀뒤, 이를 목적지 주소에 저장한다.
- **SUB** 목적지 피연산자에서 소스 피연산자를 빼고 결과를 목적지 주소에 저장한다.
- **XOR** 목적지 피연산자와 소스 피연산자에 대해 XOR 연산을 수행하고 목적지 주소에 저장한다.
- **XADD** Exchange and ADD를 뜻하며, 목적지 피연산자와 소스 피연산자의 값을 교체하고, 목적지 피연산자에 두 값의 합을 저장한다.
- **XCHG** Exchange Register/Memory with Register를 뜻하며, 목적지 피연산자와 소스 피연산자의 내용을 교체한다.

또한, Memory Barrier와 관련된 명령어들도 존재하며, 이는 아래와 같다.

- **LFENCE** 이 명령어 앞에 있는 모든 메모리로부터의 로드 명령어에 대해 직렬화를 수행한다.
- **MFENCE** 이 명령어 앞에 있는 모든 메모리로부터 로드 명령어와 메모리로의 저장 명령어에 대해 직렬화를 수행한다.
- **SFENCE** 이 명령어 앞에 있는 모든 메모리로의 저장 명령어에 대해 직렬화를 수행한다.

각 명령어를 활용한 동기화 방식은 다음 장에서 Background로 다룬다.

2 Background

2.1 Fetch and Add

Fetch and add는 이름에서 뜻하듯, $x = x + a$를 원자적으로 처리한다는 것을 뜻한다. 순서를 단계적으로 표현하면 다음과 같다.

1. 주소 x로부터 값 x_{old}을 읽어서 레지스터에 가져온다.
2. a를 레지스터 내의 x_{old}에 a를 더한다.
3. 위의 합 x_{new}를 주소 x에 저장한다.

2.2 Compare and Swap

Compare and swap은 특정 주소의 값이 특정 값과 같으면 새로운 값으로 치환하는 연산이다. 순서를 단계적으로 표현하면 다음과 같다.

1. 주소 x로부터 값 x_{val}을 읽는다.
2. 읽은 값 x_{val}을 x_{old}와 비교한다.
3. 위 두 값이 동일하면 주소 x에 x_{new} 값을 저장한다.
4. 위 두 값이 다르면, 아무 일도 수행하지 않는다.

2.3 Memory Barrier

특정 연산의 순서를 강제하도록 하는 기능으로, 로드 명령어에 대해서 직렬화를 수행하는 LFENCE, 저장 명령어에 대해서 직렬화를 수행하는 SFENCE, 모든 메모리 관련 명령어에 대해 수행하는 MFENCE가 있다. 이 기능을 통해서 CPU가 배리어로 막혀있는 두 명령의 순서를 바꿀 수 없도록 할 수 있다.

3 Experiment

3.1 실험 내용

다중 쓰레드의 동기화 작업으로 인한 직렬화와의 비교를 수행한다. 실험 수행 내용은 다음과 같다. 1,000,000,000까지의 counter를 아래 조건에 따라 10번씩 수행하여 각각의 평균과 표준편차를 구해보았다.

1. fetch_and_add, compare_and_swap, test_and_set으로 구현한 카운터에 따른 성능 비교
2. 쓰레드 개수 1개, 2개, 4개, 8개에 따른 성능 비교

3.2 실험 소스

구현 언어는 C로 수행하였으며, fetch_and_add와 compare_and_swap은 assembly 코드로 x86_64에 맞게 구현하였으며, test_and_set은 GCC의 built-in 함수를 사용하여 구현하였다.

3.2.1 공통 헤더

```
1  #include <time.h>
2  #include <sys/time.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <pthread.h>
6  #define COUNT 1000000000
7  #define NUM_THREADS 2
8
9  unsigned long get_current_microseconds()
10 {
11    struct timeval curr;
12    gettimeofday(&curr, NULL);
13
14    return curr.tv_sec * 1000000 + curr.tv_usec;
15 }
```

공통 헤더는 성능 측정을 위한 기본 정의와 측정 함수를 담았다. 6행에서는 카운터의 상한을 정의하였으며, 7행은 쓰레드의 개수를 정의하였다. 여기서 쓰레드의 개수는 파라미터로써, 1개, 2개, 4개, 8개로 변화시켜 가면서 재컴파일하여 실험을 수행하였다.

3.2.2 기본 뼈대

```
1  #include "increment.h"
2
3  FILE *fp;
4  unsigned long count;
5  volatile int lock = 0;
6
7  void *increment(void *);
8
9  pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
10
11 int main(int argc, char **argv)
```

```
12  {
13     if (argc != 2)
14     {
15        printf("Multi thread increment with the synchronization method\n");
16        printf("Please insert the result file name\n");
17        printf("USAGE: %s <result file name>\n", argv[0]);
18        exit(1);
19     }
20
21     printf("Start the multi thread increment with the fetch and add\n");
22
23     fp = fopen(argv[1], "w");
24     fprintf(fp, "Round, Value, Time\n");
25
26     pthread_t thread[NUM_THREADS];
27     pthread_attr_t attr;
28     pthread_attr_init(&attr);
29     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
30
31     int t, rc, round = 0;
32     void *status;
33     unsigned long start, end;
34
35     while (round < 10)
36     {
37        start = get_current_microseconds();
38        for (t=0; t<NUM_THREADS; t++)
39        {
40           rc = pthread_create(&thread[t], &attr, increment, NULL);
41
42           if (rc)
43           {
44              printf("ERROR: return code from pthread_create: %d\n", rc);
45              return 1;
46           }
47        }
48
49        pthread_attr_destroy(&attr);
50        for (t=0; t<NUM_THREADS; t++)
51        {
52           rc = pthread_join(thread[t], &status);
53           if (rc)
54           {
55              printf("ERROR: return code from pthread_join: %d\n", rc);
56              return 1;
57           }
58        }
59        end = get_current_microseconds();
60
61        fprintf(fp, "%d, %ld, %ld\n", round, end - start, count);
62        printf("Round %d\n", round);
63        printf("Elapsed Time: %ld us\n", end - start);
64        printf("Final Value: %ld\n", count);
65
66        round++;
67        count = 0;
68     }
69     pthread_exit(NULL);
70     close(fp);
71
72     return 0;
```

```
73 }
```

4행에서 공통 변수로 unsigned long 타입의 count를 선언하였으며, 쓰레드들은 이 변수로의 접근을 두고 경쟁하게 된다.13-19행은 오류 처리 구문이며, 21행은 정상 동작 표시이다. 23-24행은 결과를 출력하기 위한 파일을 초기화한 것이고, 26-29행은 쓰레드를 초기화한 구문이다. 37-59행이 실험 한 번 수행에 해당하며 61-64행이 결과 출력 구문이다. 66-67행은 다음 실험 수행을 위한 초기화 구문이다. 이를 35행에서 보듯 10회 반복한다.

3.2.3 fetch_and_add

1. fetch_and_add 함수

```
1 static inline int fetch_and_add(unsigned long *ptr, unsigned long val)
2 {
3    __asm__ volatile("lock; xadd %0, %1"
4       : "+r" (val), "+m" (*ptr)
5       :
6       : "memory"
7    );
8    return val;
9 }
```

fetch_and_add 함수는 ptr가 가리키는 주소 값을 읽어와서 val을 더한 뒤 해당 주소에 다시 저장하는 것이다. Assembly 코드에서 확인 가능하듯이 xadd 수행 전에 lock을 수행하여 동기화를 수행한다.

2. fetch_and_add 쓰레드 수행

```
1 void *increment(void *p)
2 {
3    unsigned long tmp;
4    int i;
5    while (count < COUNT)
6    {
7       fetch_and_add(&count, 1);
8    }
9 }
```

공통 변수인 count가 10억(COUNT)보다 작을 때 fetch_and_add를 수행한다. 이 코드는 문제가 있는데, 여러 쓰레드가 조건을 충족하여 fetch_and_add 함수 직전에서 대기할 수 있다. 이로인해 이 함수의 최종 결과는 count를 [10억, 10억+(# of threads -1)] 범위 내에 있도록 만든다.

3.2.4 compare_and_swap

1. compare_and_swap 함수

```
1  static inline unsigned long compare_and_swap(unsigned long *ptr, unsigned long
       old_val, unsigned long new_val)
2  {
3     unsigned long ret = 1;
4
5     __asm__ volatile(
6        "movq %2, %%rax\n\t"
7        "movq %3, %%rbx\n\t"
8        "movq %0, %%rcx\n\t"
9        "lock\n\t"
10       "cmpxchg %%rbx, (%%rcx)\n\t"
11       "jz done\n\t"
12       "movq $0, %1\n\t"
13       "done: \n\t"
```

```
14          :"=m"(ptr), "=g"(ret)
15          :"g"(old_val), "g"(new_val), "m"(ptr)
16          :"%rax", "%rbx", "rcx", "cc"
17      );
18
19      if (ret)
20          return 0;
21  }
```

10행에서 *ptr과 old_val을 비교하며, 만약 동일한 값이면 new_val이 저장되면서 ZF(Zero Flag)가 설정된다. 11행에서 만약 ZF가 설정되면 종료되고, 그렇지 않으면 ZF가 초기화되고 본래 값이 유지된다.

2. compare_and_swap 쓰레드 수행

```
1   void *increment(void *p)
2   {
3       unsigned long tmp;
4       while (count < COUNT)
5       {
6           tmp = count;
7           if (tmp < COUNT)
8               compare_and_swap(&count, tmp, (tmp + 1));
9       }
10  }
```

카운터는 우선 4행에 진입하면서 공통 변수 count가 10억(COUNT)보다 작은지 확인하고 6행에서 tmp 변수에 현재 count를 저장한다. 만약 tmp가 10억보다 작으면 compare_and_swap 함수에 진입하며, 그 사이에 다른 쓰레드가 count를 1 증가시키지 않았다면, 즉, count 값과 tmp 값이 동일하다면, 8행에서 tmp에 1 증가시킨 값을 count에 대입한다.

3.2.5 test_and_set

1. test_and_set 쓰레드 수행

```
1   void *increment(void *p)
2   {
3       unsigned long tmp;
4       int lock;
5       while (count < COUNT)
6       {
7           while(__sync_lock_test_and_set(&lock, 1))
8           {
9               if (count < COUNT)
10              {
11                  count++;
12                  __sync_lock_release(&lock);
13              }
14          }
15      }
16  }
```

test_and_set 함수는 보통 7행 처럼 lock을 사용할 때 사용하며, 내부적으로는 BTS(Bit Test and Set)를 활용하여 수행된다. 쓰레드는 5행에서 공통 변수 count가 10억(COUNT)보다 작은 지 확인한 후, 7행에서 lock을 수행하며, 그 사이 5행과 7행 사이에 count가 증가될수도 있으므로 9행에서 count가 10억보다 작은지 다시 확인이 되면 11행에서 count를 1 증가시킨다. 그리고 나서 12행에서 lock을 해제한다.

3.3 실험 결과 및 분석

구현 및 쓰레드 개수에 따른 수행 시간 측정 결과는 아래 그래프1와 같다. 각각의 값은 10회 수행에 대한 평균값이다. 쓰레드 8개일 때 Test and Set는 수행이 되지 않아 측정 결과에서 제외하였다. 그 밑의 그래프 2는 동일한 상황 하에서 표준 편차에 해당한다.

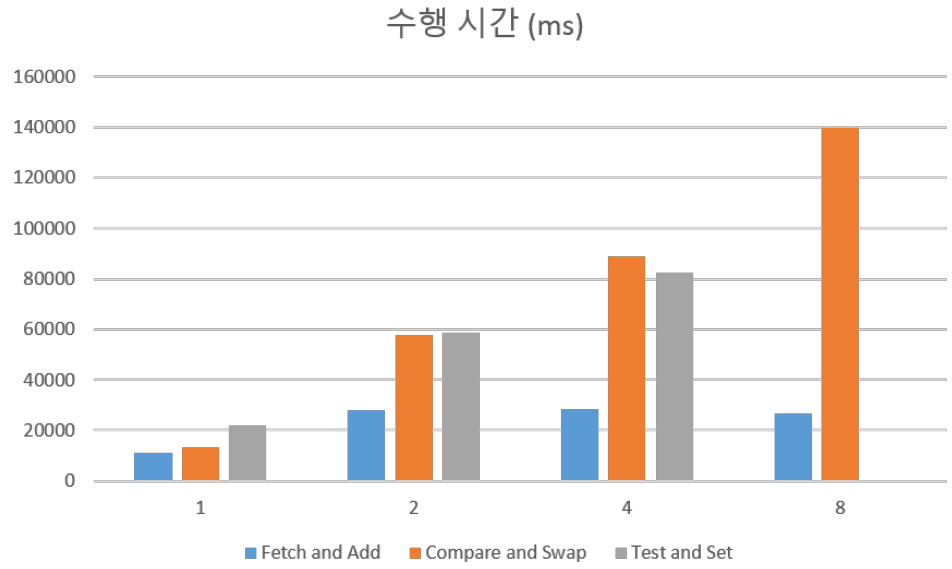

Figure 1: 각 구현 및 쓰레드 개수에 따른 수행 시간 비교 (단위: ms)

수행 시간을 살펴보면, Fetch and Add가 다른 두 가지에 비해 짧은 시간 동안 수행된 것을 확인할 수 있으며, 쓰레드 2개 이상에서 비슷한 정도의 시간이 걸린 것을 확인할 수 있다. 반대로, Compare and Swap이나 Test and Set은 쓰레드 개수에 비례하여 수행 시간이 증가한 것을 확인할 수 있다.

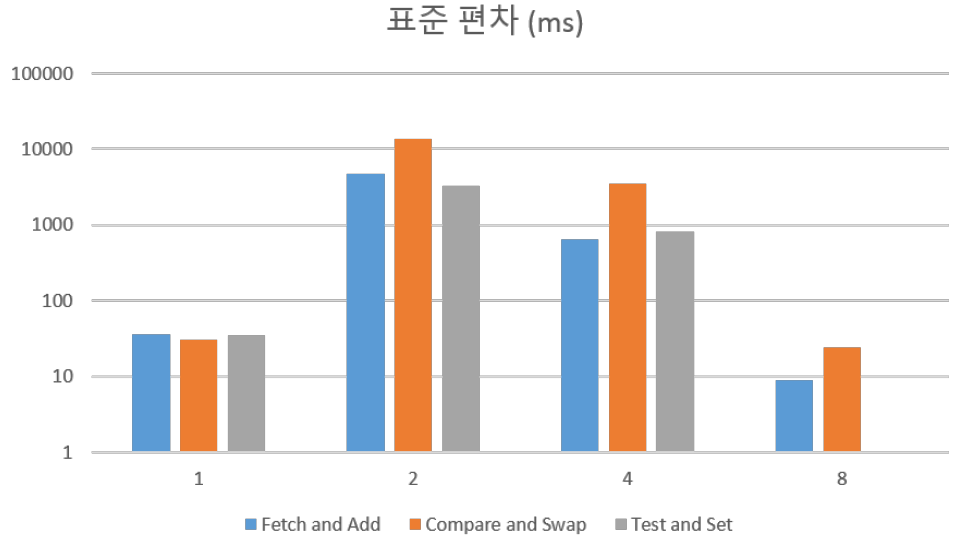

Figure 2: 각 구현 및 쓰레드 개수에 따른 표준 편차 비교 (단위: ms, 세로축 로그 스케일)

표준 편차를 살펴보면, 3가지 모두 유사한 경향을 보였다. 쓰레드 두 개가 수행될 때, 표준 편차가 극점을 찍고, 쓰레드가 증가할수록 표준 편차가 감소하는 것을 확인할 수 있었다.

하나의 공유 변수에 대해 여러 쓰레드가 한 번에 1씩 증가시키는 카운터의 특성 상 병렬화가 큰 의미 없이 작용하기 때문에, 쓰레드의 개수가 증가하면 싱글 쓰레드에 비해 수행 시간이 증가한다는 것을

확인할 수 있었다. 이는 두 개 이상의 쓰레드 수행 상황에서 lock에 의한 대기 시간이 발생하기 때문이라 볼 수 있다. 다만, Fetch and Add의 경우, Compare and Swap이나 Test and Set과 달리 두 개 이상의 쓰레드에서 대체로 일정한 수행 시간이 발견되는 것을 확인할 수 있는데, 이는 다른 두 개의 연산에 비해 가볍기 때문으로 사료된다. 반대로, 두 개의 연산이 무거운 까닭은 Compare and Swap과 Test and Set은 내부적으로 두 피연산자의 값을 교체하는 과정 때문으로 생각된다.

표준 편차를 살펴보면, 두 개의 쓰레드가 수행하고 있을 때, 가장 높은 값을 보이는 것을 알 수 있다. 그 이유는 lock이 해제되었을 때 경쟁을 수행하여 특정 쓰레드가 Critical Section에 들어갈 확률이 낮아지는 4개나 8개의 경우와 달리, 2개의 경우는 상대가 lock을 해제하면 바로 다른 쓰레드가 Critical Section에 진입하기 때문이다. 따라서 이 경우, 다른 쓰레드의 수행 시간이 분산하는 만큼 다른 쓰레드의 대기 시간도 분산한다.

4 소결

본 과제에서는 다중 쓰레드 상황에서 동기화를 수행하기 위한 연산을 살펴보고 10억까지의 카운터를 통해 각각의 성능을 살펴보았다. x86_64 아키텍처가 제공하는 명령어들을 살펴보고, 이를 바탕으로 Fetch and Add와 Compare and Swap, 그리고 Test and Set을 구현하여 쓰레드의 개수를 변화시키면서 10회 수행 시간을 측정하고 각각의 평균과 표준편차를 살펴보았다. 공통 변수 한 개의 값을 증가시키는 연산이기 때문에 결과적으로 직렬화가 되어 병렬화가 큰 효과가 없는 프로세스였다. Fetch and Add는 다른 두 개의 연산에 비해 빠른 수행 시간을 보였으며, 이는 다른 두 개 연산에 두 피연산자의 값을 교체하는 부분이 무겁기 때문으로 보인다. 표준 편차는 두 개의 쓰레드에서 가장 높은 값을 보였으며, 이는 쓰레드 두 개 상호 불확실성의 결과라고 볼 수 있다.

References

[1] Guide, P. Intel® 64 and ia-32 architectures software developer's manual. *Volume 2: Instruction Set Reference* (2011).