Lock-based Concurrent DS

20191562 김성훈

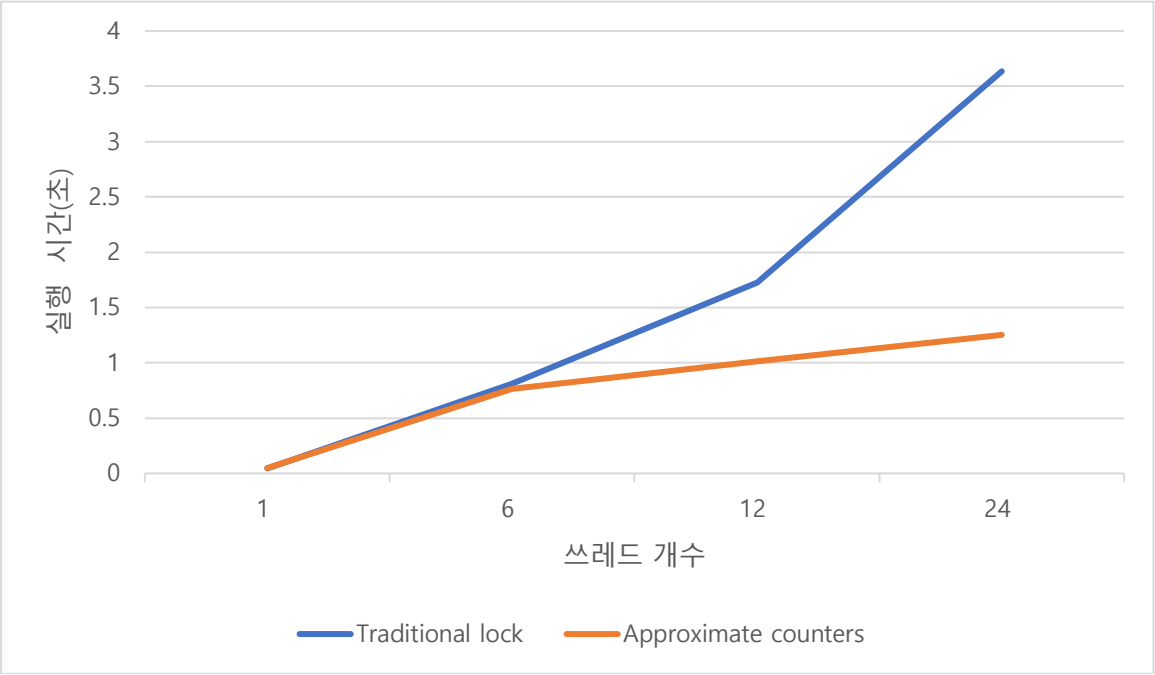
실험환경 분석

물리적인 CPU 코어 개수: 6
하이퍼쓰레딩을 적용해서 실제로 동시에 처리할 수 있는 쓰레드 개수: 12

Performance of Traditional vs. Approximate Counters

로컬 PC 의 환경을 고려해 테스트 threads 개수는 1, 6, 12, 24 로 진행한다.
Traditional lock 은 global lock 을 이용해서 모든 threads 가 하나의 lock 만 사용하고 Approximate counters 의 lock 은 각각의 threads 가 local lock 을 보유하며 threshold 에 도달할 때마다 global lock 을 이용한다.

비교 결과를 먼저 본 후 실제 연산하는 부분의 어떠한 차이로 인해 성능이 비교되는지 확인하겠다.



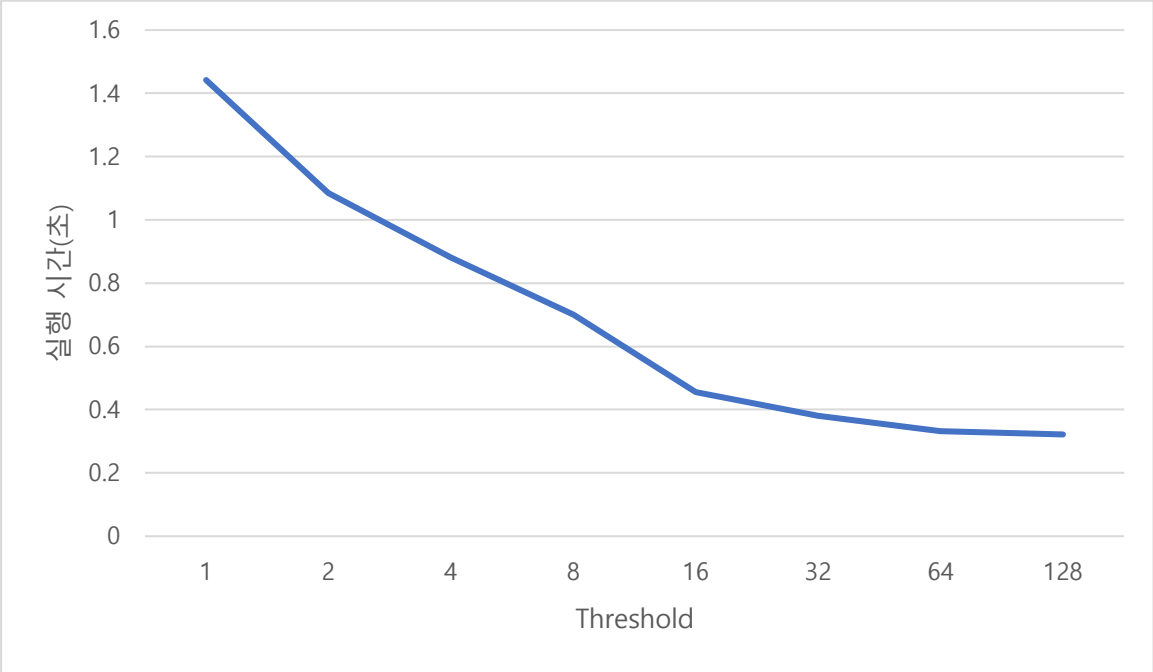
Traditional lock 은 포함된 코드 중
- counter_with_lock_thread_1.c
- counter_with_lock_thread_6.c
- counter_with_lock_thread_12.c
- counter_with_lock_thread_24.c
를 실행했고
Approximate counters 는 Threshold 를 1024 로 설정하고
- approximate_counter_1.c
- approximate_counter_6.c
- approximate_counter_12.c
- approximate_counter_24.c
를 실행했다.

Traditional lock 은 기울기가 therads 개수가 12 를 넘어가고 난 후부터 급격하게 변화하고 그에 비해 Approximate counters 는 별로 차이가 나지 않는다.
실험 환경은 cpu core 가 12 개이기 때문에 한 번에 12 threads 를 처리 가능한데 lock 으로 인해 1 thread 만 처리할 수 있게됐다.
여기서 12 개의 처리를 1 사이클로 가정하면 1 사이클안에 최대 12 threads 가 처리 가능하기 때문에 12 threads 까지의 쓰레드 테스트 기울기는 별로 차이가 나지 않는 것을 확인할 수 있다. 그런데 12 를 초과하게 되면 사이클의 제한을 초과하는 것과 동일하기 때문에 초과한 개수만큼 한 사이클이 끝날 때 밀리게 된다. 이는 24 threads 테스트 시 한 사이클이 끝나면 12 개의 처리 연산 즉, 한 사이클이 추가로 필요하다는 것을 의미한다. 이를 실제 실행 시간을 확인해서 검증해보겠다.
12 threads 의 실행 시간은 1.725696s 이며 24 threads 의 실행 시간은 3.634974s 이다.
대략적으로 차이가 2 배이니 다른 overheads 들을 고려한다면 위에서 설명한 내용이 성립한다.
여기서 overheads 는 테스트를 위한 프로세스가 아닌 인터넷 브라우저와 같은 불필요한 프로세스의 실행을 위한 context switch 같은 것을 의미한다.

Approximate counters 는 Threshold 가 1024 로 설정됐기 때문에 각각의 Thread 내부의 lock 을 항상 실행하고 global lock 은 1024 번 마다 실행한다. 그렇기 때문에 한 사이클을 여기선 1024 라고 볼 수 있다. 만약 Threads 개수가 1024 와 2048 로 테스트 됐다면 급격한 변화가 Traditional lock 과 유사하다. 실제로 수치를 약간 변경하고 threads 개수만 조절한다면 1.973917s 와 3.882913s 의 결과가 나온다.

Approximate Counters

이전에 Approximate counters 의 threads 변화에 따른 성능을 살펴봤다면 이번엔 threshold 에 따른 성능 변화를 살펴보겠다.
최대한 다른 영향을 받지 않기 위해 Threads 개수는 12 로 설정하고 테스트를 진행했다.

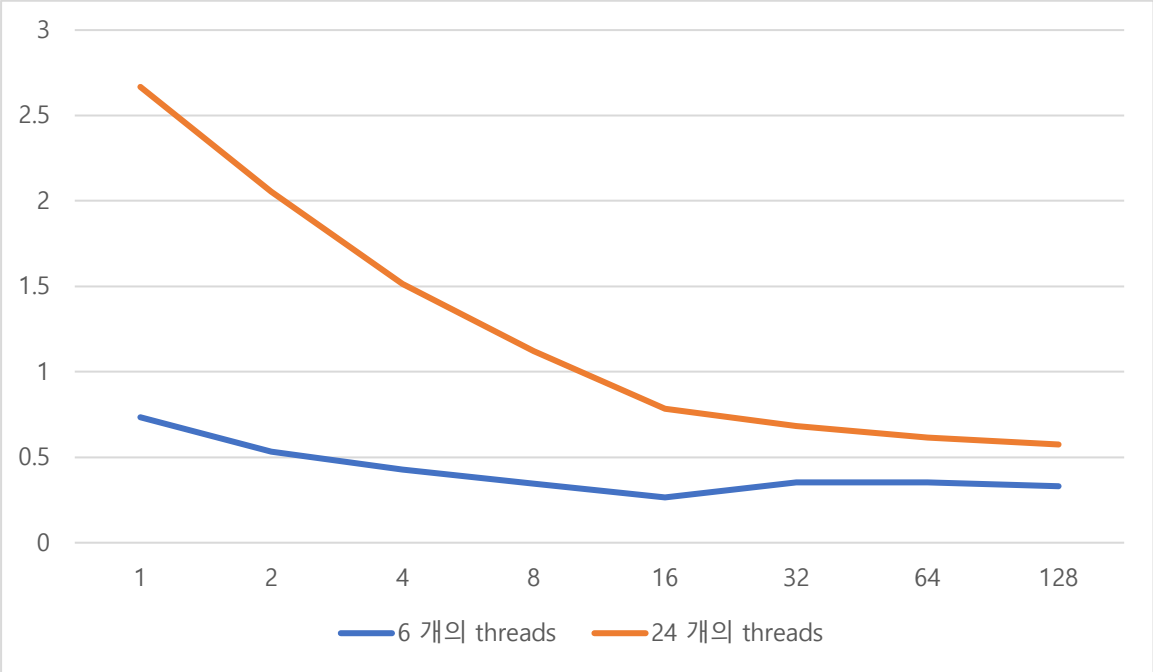


위 결과는 approximate_counter_12_convergence.c 를 실행해서 측정됐다.

```c
void update(counter_t *c, int threadID, int amt)
{
  int cpu = threadID;
  pthread_mutex_lock(&c->llock[cpu]);
  c->local[cpu] += amt; // 여기서는 amount 가 1 로 고정된다.
  if (c->local[cpu] >= c->threshold)
  {
    pthread_mutex_lock(&c->glock);
    c->global += c->local[cpu];
    pthread_mutex_unlock(&c->glock);
    c->local[cpu] = 0;
  }
  pthread_mutex_unlock(&c->llock[cpu]);
}
```

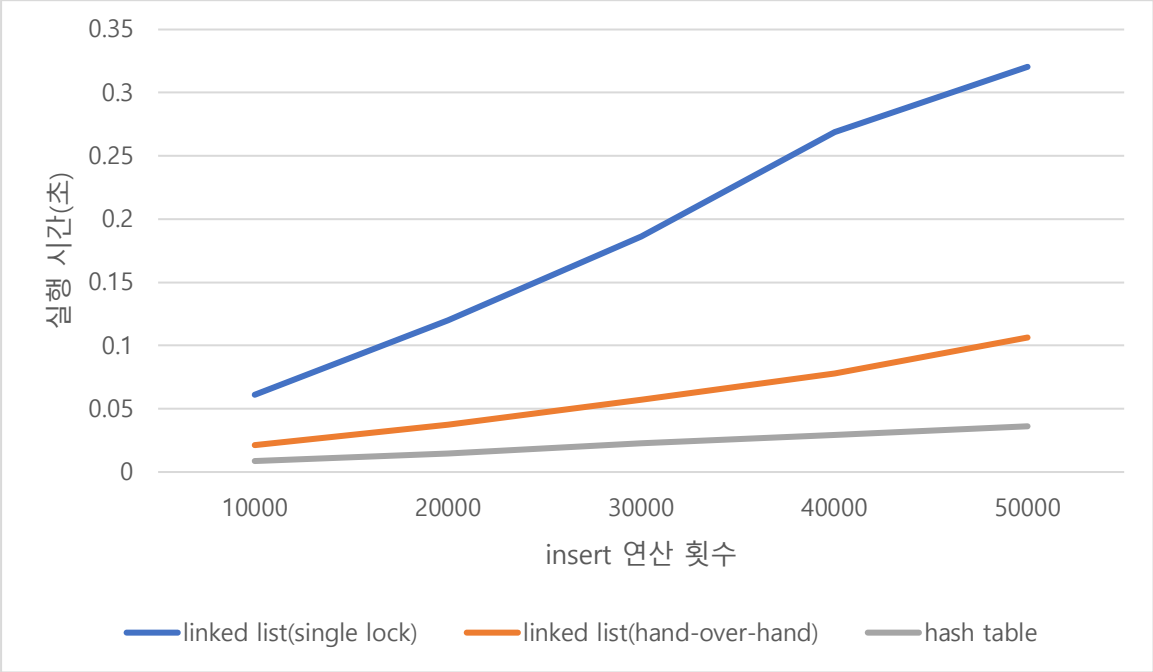
위 코드가 핵심 부분인데 코드의 분석을 통하면 threads 개수가 6, 24 일 때는 어떤 결과가 나오는지 예측할 수 있다. 이는 현재 12 로 설정한 결과가 왜 나온 것인지도 이해할 수 있게 도와준다.

로컬에 저장된 값이 threshold 에 도달하면 global lock 을 획득하고 실제 값을 변경하고 lock 을 반환하게 된다. 이는 간단하게도 threads 개수가 threshold 보다 크다면 bottleneck 이 발생하는 것을 의미한다.
그렇다면 threads 개수가 threshold 보다 높다면 bottleneck 이 발생한다는 것을 의미한다. 이는 차트를 보면 분명하게 알 수 있기 때문에 넘어가겠다.
또한 6, 24 의 threads 개수로 실행한 결과를 차트를 통해 간단히 보고 넘어가겠다.



Performance of Data Structures

카운트를 관리하는 방법의 성능을 이전에 살펴봤다면 여기서는 일반적으로 사용하는 linked-list 와 hash table 에 race condition 을 없애기 위한 방식과 성능을 살펴보겠다.
hash table 은 간단하게도 conflict 가 발생하는 경우만 ciritical section 을 각각의 lock 이용하고 아니면 병렬적으로 처리한다.
linked list 는 hand-over-hand 방식과 single lock 이용해서 관리하는 두 가지 방식이 있는데 차트를 먼저 보고나서 설명하겠다.



위 결과는 복잡한 상황을 연출하기 위해 0~99 구간의 index 에 insert 를 진행했다. 또한 threads 개수는 12 이다.

hash table 은 bucket 의 개수에 따라 성능이 결정된다. 여기서는 bucket 을 101 개로 설정했기 때문에 threads 와는 많이 차이난다. 이는 단순히 insert 연산 횟수에 성능이 결정된다는 것을 의미한다. 만약 threads 가 202 개라면 threshold 처럼 기울기가 급격히 변화할 것이다.

hash table 은 항상 빠른 것을 알았으니 single lock 과 hand-over-hand 방식을 비교할 것인데 간단히 hand-over-hand 방식을 설명하겠다.
1. 현재 위치의 lock 을 획득한다.
2. 다음 위치의 lock 을 획득한다.
3. 다음 위치로 넘어간다.
4. (1) 에서 획득한 lock 을 반환한다.
1 ~ 4 를 반복하며 목표 위치까지 이동한다.

```
pthread_mutex_t f_lock = L->head->lock;
pthread_mutex_t s_lock;
pthread_mutex_lock(&f_lock);
node_t *cur = L->head;
for (int i = 0; i < index; i++)
{
  s_lock = cur->next->lock;
  pthread_mutex_lock(&s_lock);
  cur = cur->next;
  pthread_mutex_unlock(&f_lock);
  f_lock = s_lock;
}
new->next = cur->next;
cur->next = new;
pthread_mutex_unlock(&f_lock);
```

실제 구현하게 된다면 코드는 위와 같은 형식일 것이다. 또한 실제로 테스트 시 위 코드를 사용했다.

hand-over-hand 방식을 알았으니 single lock 을 살펴보겠다.

```
pthread_mutex_lock(&L->lock);
  node_t *cur = L->head;
  for (int i = 0; i < index; i++)
  {
    cur = cur->next;
  }
  new->next = cur->next;
  cur->next = new;
  pthread_mutex_unlock(&L->lock);
```

단순히 하나의 lock 만 사용하기 때문에 코드가 간결하고 list 내부의 한 개인 lock 를 이용하는 것이 직관적이다.

동작을 살펴봤으니 실제 성능 차이가 왜 발생하는지 알아보겠다.
0 ~ 99 구간의 index 를 돌아가며 insert 하는 것을 이전에 설명했다. 이는 single lock 병렬적으로 처리할 때 처리해야 될 index 가 높을 수록 성능을 하락시킨다는 것을 테스트하기 위함이었다.
0 번 째를 lock 해야 된다는 점에서 hand-over-hand 방식과 동일하지만 만약 99 번 째를 생각한다면 결과가 완전히 다를 것이다.
hand-over-hand 는 동시에 최대 2 개의 lock 을 사용하기 때문에 99 index 까지 traversal 한다면 50 개의 threads 가 실행 가능한데 single lock 은 한 개만 실행 가능하기 때문이다.
테스트 데이터는 동일하기 때문에 차이를 확인함으로써 위 내용이 맞는 것을 알 수 있다.