Distributed System Homework1-2 Report

2017-11394 김형모

1 Synchronization Operations

대표적인 Synchronization Operation에는 `Test-and-Set`, `Compare-and-Swap`, `Fetch-and-Add`, `Memory barrier` 등이 있다. martini[1]에서는 `x86_64 architecture`에서 제공하는 instruction을 지원하는데, `Test-and-Set`에 대응하는 것으로는 **BTS** Instruction이, `Compare-and-Swap`에 대응하는 것으로는 **CMPXCHG** Instruction이, `Fetch-and-Add`에 대응하는 것으로는 **ADD** 또는 **XADD** Instruction이 있다. 이들 Instruction은 **LOCK** Prefix를 붙여 atomic operation으로 바꿀 수 있다.

2 Parallel programming model

본 과제는 각 thread가 공유하는 변수 `mem`에 10억번 1을 더하는 연산을 수행하는 것이다. 이를 1개의 코어에서 실행한 Serial code는 다음과 같다. 이때 `mem`에 10억번 1을 더하는데 1초가 걸렸다.

```
while (mem < COUNT) {
  mem++;
}
```

`pthread`를 이용하여 구현한 Parallel code는 다음과 같다.

```
long long int i;
for (i = 0; i < TNUM; i++) {
  pthread_create(&tid[i], NULL, work, (void *) i);
}

for (i = 0; i < TNUM; i++) {
  void * ret;
  pthread_join(tid[i], &ret);
}
```

[1] 컴퓨터공학부 숙제 서버

3 Test-and-Set

BTS Instruction으로 다음과 같이 work thread를 구현하였다.

```
void * work_bts(void * vargp) {
  int count = COUNT;

  while (mem < COUNT) {
    __asm__ __volatile__(
      "L_aquire%=:" ENDL
      LOCK_PREFIX "bts $0, %0" ENDL
      "jc L_aquire%=" ENDL
      "movl %1, %%eax" ENDL
      "cmpl %2, %%eax" ENDL
      "jge L_release%=" ENDL
      "incl %1" ENDL
      "L_release%=:" ENDL
      "movl $0, %0" ENDL
      : "+m"(lock), "+m"(mem), "+m"(count)
    );
  }

  return NULL;
}
```

각 thread는 `lock`이 0이면 이를 1로 바꾸고 critical-section으로 진행하고, `lock`이 1이면 `spinlock`에 걸리게 된다. **BTS** Instruction에 **LOCK** Prefix를 붙여 `mutex`를 atomic하게 얻도록 하였다. 이러한 방법으로 `mem`에 10억번 1을 더하는데 총 556초가 걸렸다.

4 Compare-and-Swap

CMPXCHG Instruction으로 다음과 같이 work thread를 구현하였다.

각 thread는 `mem`을 `load`한 후, `mem`의 기존 값에 변화가 없으면 그보다 1만큼 큰 값으로 교체한다. 만약 다른 thread가 먼저 값을 변경한 경우 재시도하게 된다. **CMPXCHG** Instruction에 **LOCK** Prefix를 붙여 비교를 atomic하게 하도록 하였다. 이러한 방법으로 `mem`에 10억번 1을 더하는데 총 166초가 걸렸다.

5 Fetch-and-Add

ADD Instruction으로 다음과 같이 work thread를 구현하였다.

각 thread는 `vargp`로부터 자신이 더해야하는 총량을 계산하고, 그만큼 `mem`에 1을 더한다. **Add** Instruction에 **LOCK** Prefix를 붙여 값을 `load`하여 `add`

```
void * work_cas(void * vargp) {
  int count = COUNT;

  while (mem < COUNT) {
    __asm__ __volatile__(
      "L_loop%=:" ENDL
      "movl %0, %%eax" ENDL
      "cmpl %1, %%eax" ENDL
      "jge L_done%=" ENDL
      "movl %%eax, %%edx" ENDL
      "incl %%edx" ENDL
      LOCK_PREFIX "cmpxchgl %%edx, %0" ENDL
      "jz L_loop%=" ENDL
      "L_done%=:" ENDL
      : "+m"(mem), "+m"(count)
    );
  }

  return NULL;
}
```

```
void * work_add(void * vargp) {
  int i = *((int *) &vargp);
  int load = (COUNT / TNUM) + (i < (COUNT % TNUM) ? 1 : 0);

  while (load--) {
    __asm__ __volatile__(
      LOCK_PREFIX "addl $1, %0" ENDL
      : "+m"(mem)
    );
  }

  return NULL;
}
```

하는 전 과정을 atomic하게 하도록 하였다. 이러한 방법으로 `mem`에 10억번 1을 더하는데 총 23초가 걸렸다.

6 Conclusion

실험 결과, 동기화에서 software logic의 비중이 클수록 성능이 떨어졌다. 동기화를 위해 필요한 부가 연산(`compare`, `branch`)이 많아지고, spinlock과 같은 비효율적인 알고리즘을 사용했기 때문으로 보인다.[2]

[2]본 과제에서는 counter를 구현하는 것이 목적이므로, `Memory barrier`는 이용하지 않았다.