Wait-free 동기화 및 메모리 해제 기술 연구*

신은환, 김인혁, 엄영익
성균관대학교 정보통신공학부
{comsky, kkojiband, yieom}@ece.skku.ac.kr

A Study on Wait-Free Synchronization and Memory Reclamation Technologies

Eun Hwan Shin, Inhyuk Kim, Young Ik Eom
School of Information and Communication Engineering
Sungkyunkwan University

요 약

Locking은 다중 스레드간의 경쟁 상태를 조절하기 위한 전통적인 메커니즘이다. 하지만 Locking을 사용할 경우, 공유 데이터에 대한 잠금(lock) 및 해제(unlock)에 따른 대기 시간(waiting time)이 발생하며, 이는 전체 시스템 성능을 저하시킨다. Wait-free 동기화는 이러한 전통적인 Locking의 비용을 줄이고자 하는 기법이다. Wait-free 동기화의 기본 아이디어는 공유 데이터 수정 시 복제본을 생성해 처리함으로써 잠금에 따른 대기시간을 제거하는 것이다. 따라서 Wait-free 동기화 기법에서는 복제본 생성 이후의 메모리 해제가 가장 큰 비용을 차지한다. 이에 본 논문에서는 Wait-free 동기화 및 메모리 해제 기법과 관련하여 주요 이슈 및 기술 현황에 대한 분석을 실시하였다.

1. 서론

마이크로프로세서의 최대 성능을 이끌어내는 것은 소프트웨어 개발자들에게 있어 중요한 이슈가 되어왔다. 하지만 최근 싱글코어 프로세서의 성능 향상은 한계에 부딪쳤으며, 복수개의 코어를 사용하는 멀티코어 방식의 프로세서가 새로운 대안으로 떠오르고 있다. 그림 1은 미국 MIT의 Amarasinghe 교수에 의해 조사된 지난 20여 년간의 프로세서 성능 발전 동향이다. 그래프를 살펴보면 2004년 이후로 프로세서의 절대적인 성능, 즉 코어 주파수 성능의 향상은 거의 이루어지지 않고 있다는 것을 알 수 있으며, 이는 하드웨어 설계상의 한계, 지나친 전력 소모 등 다양한 요인에 기인한다. 이에 관련 업계에서는 코어 성능의 한계를 극복하기 위해 다양한 멀티코어 프로세서를 출시하고 있다. 이와 같은 멀티코어 프로세서는 성능의 개선과 더불어 복수개의 코어를 활용한 작업의 병렬처리를 가능하게 하였으며, 이는 생산성의 향상으로 이어지고 있다. 하지만 병렬처리 개념의 도입은 공유 데이터 처리와 같은 새로운 문제들을 발생시키며, 동시 접근에 대한 효과적인 조정, 멀티스레드 프로그래밍 등이 소프트웨어 개발자들에게 새로운 도전과제가 되고 있다.

Locking은 다중 스레드간의 경쟁 상태를 조절하기 위

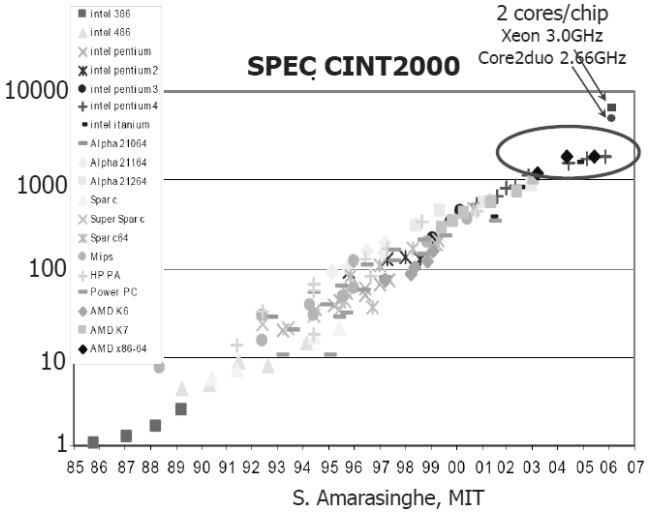

그림 1 프로세서 성능 발전 동향

한 전통적인 메커니즘이다. 하지만 이는 데이터에 대한 잠금(lock) 설정 후 해당 데이터가 잠금 해제(unlock)되기 위한 대기 시간(waiting time)을 발생시키며, 이는 전체 시스템 성능을 저하시킨다.

Wait-free 동기화는 이러한 전통적인 Locking의 비용을 줄이고자 하는 기법이다. 'Wait-free'는 모든 동작이 교착상태(deadlock)에 빠지지 않고 범위(boundary) 내에 완료되는 것을 보장한다는 의미이다. Wait-free 동기화의 기본 아이디어는 수정하고자 하는 공유 데이터의 복제본을 생성해 처리한 뒤, 처리된 결과를 공유 데이터와 동기화

* 본 연구는 지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 육성지원사업의 연구결과로 수행되었음 (NIPA-2011-(C1090-1121-0008))

하는 것이다. Wait-free 동기화는 큰 의미에서 Lock-free의 개념을 포함하고 있으며, 따라서 Locking을 사용하지 않기 때문에 복제본 생성시 대기 시간이 없는 메모리 할당을 수행한다. 이로 인해 Wait-free 동기화 기법에서는 할당된 메모리에 대한 해제가 가장 큰 비용을 차지한다.

이에 본 논문에서는 Wait-free 동기화 및 메모리 해제 기법 관련기술을 분석한다. 본 논문의 2장과 3장에서는 Wait-free 동기화 및 메모리 해제 기법과 관련하여 주요 이슈 및 기술 현황에 대해 분석을 실시한다. 마지막으로 4장에서 본 논문을 마무리한다[1].

2. Wait-free 동기화 기술

2.1 RCU(Read-Copy Update)

RCU는 복수개의 프로세스(또는 스레드)에 의해 동시에 수정되고 있는 공유 데이터에 대한 고성능의 읽기 연산을 제공하는 Lock-free 동기화 기법이다. 1998년 PDCS(Parallel and Distributed Computing and Systems)에서 IBM Beaverton 연구소의 Paul E. McKenney와 John D. Slingwine에 의해 제안되었으며, 리눅스 커널 2.5.43 버전부터 공식 패치에 포함되었다. 아래 그림 2는 패치 적용 이후 현재까지 리눅스에서 RCU API와 Locking API의 사용량을 비교하여 보여주는 그래프이다. 이를 통해 RCU의 사용량이 급격한 증가 추세에 있으며, Wait-free 동기화 기법 연구에 있어 RCU가 얼마나 중요한 의미를 가지는지 알 수 있다.

RCU는 고성능의 Wait-free 읽기 연산을 제공함으로써 그 사용량이 급증하고 있지만, 이에 반해 수정된 데이터에 대한 업데이트(update) 연산에 대해서는 많은 비용을 필요로 할 수도 있다. RCU에서의 업데이트 연산의 경우 업데이트가 요청된 시점을 기준으로 기존에 공유 데이터를 참조 중인 모든 프로세스들이 연산을 완료할 때까지 기다린 후 변경된 정보를 적용하는 방법을 사용하기 때문이다.

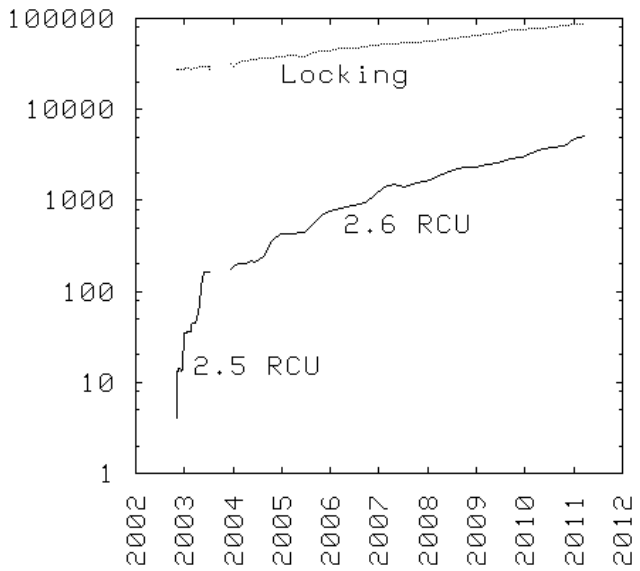

그림 2 RCU/locking API 사용량

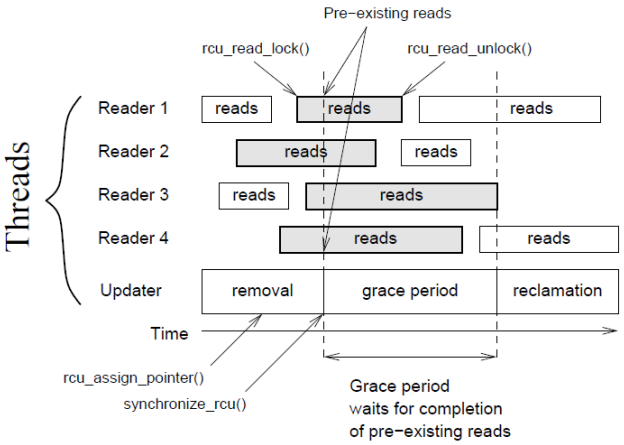

그림 3 RCU 기본 개념도

위 그림 3은 RCU 동기화 기법의 기본 아이디어를 나타내고 있다. 위 그림에서 Reader는 공유 데이터를 참조중인 프로세스를 나타내며, Updater는 공유 데이터에 대한 수정을 가하는 프로세스를 나타낸다. Reader들은 앞서 소개한 Locking과 같이 공유 데이터를 읽기 위해 잠금(lock)을 설정하고, 읽기가 완료되면 잠금 해제(unlock)를 한다. 하지만 이는 기존의 Lock 기반 동기화 기법에서처럼 공유 데이터가 잠겨 있는 동안 대기(waiting)를 수행하지 않는다. 이는 단지 읽기 연산의 시작과 끝을 나타내기 위한 명시적인 과정일 뿐이다. 따라서 RCU는 Wait-free 읽기 연산을 제공한다고 할 수 있다. 이에 반해 Updater는 공유 데이터를 수정한 뒤, `synchronize_rcu()` API에서 Grace period를 거친다. Grace period는 RCU 구현에 있어서 가장 핵심이 되는 개념으로, 업데이트가 요청된 시점에서 기존에 공유 데이터를 참조중인 모든 Reader들의 연산이 완료될 때까지의 대기 시간을 의미한다. Grace period이후에는 더 이상 공유 데이터를 참조하는 Reader가 없으므로 자원에 대한 해제가 가능하게 된다. RCU에서 각 연산의 기본 동작은 아래와 같다.

- 읽기(read) – RCU에서는 포인터를 통해서 공유 데이터에 접근이 가능하다. Reader는 `rcu_dereference()` 함수를 통해 포인터가 가리키고 있는 공유 데이터에 접근한 뒤, 해당 데이터를 읽어오게 된다.
- 수정(update) – Updater는 공유 데이터의 수정을 위해 새로운 메모리 공간을 할당받고, 그 곳의 데이터를 수정한다. Updater는 수정이 완료된 후, 기존의 포인터가 수정한 곳을 가리키도록 값을 바꿔줌으로써 동기화를 유지할 수 있게 된다. 따라서 이 후에 접근하는 Reader들은 새로 수정된 데이터를 읽게 된다. 이는 기존의 잠금(lock) 기반 동기화 기법과 가장 큰 차이점이다.
- 삭제(delete) – 모든 삭제 과정은 Grace period가 완료되었을 경우에만 수행된다. 만약 Reader가 공유 데이터에 접근하고 있는 도중에 `free()` 함수를 호출하면, 참조중인 데이터가 사라져 읽기 실패가 발생하기

때문이다. 따라서 삭제 과정의 수행을 위해서는 공유 데이터에 접근중인 Reader들을 체크할 수 있는 메커니즘이 요구된다[2].

2.2 URCU(Userspace Read-Copy Update)

URCU는 기존에 커널 레벨에서 사용되던 RCU 동기화 기법을 사용자 레벨(userspace)에서 사용할 수 있도록 구현한 것이다. URCU는 기본 API를 포함하여 리스트, 큐, 스택 자료구조를 이용하기 위한 API도 제공한다.

아래 표 1은 URCU의 기본 API를 이용한 예제 프로그램이다. main 함수에서 Reader, Writer 스레드를 생성하게 되면, 각각의 스레드가 아래의 코드를 수행하게 된다. Reader는 `rcu_register_thread()` 함수를 이용해 자신의 스레드 ID를 등록한 다음 읽기 과정을 수행한다. 여기서 `rcu_read_lock()`, `rcu_read_unlock()` 함수는 단지 읽기 과정의 수행을 알릴뿐, 실제로 공유 데이터에 대한 잠금(lock)은 수행하지 않는다. Reader 스레드에 의한 읽기 과정이 완료되면, `rcu_unregister_thread()` 함수를 이용해 Reader의 등록을 해제한다. Writer는 새로운 메모리 공간을 할당 받아 그 곳의 데이터를 수정한 다음, `rcu_xchg_pointer()` 함수를 이용해 데이터 포인터가 수정된 데이터를 가리키도록 바꿔준다. 마지막으로 `synchronize_rcu()` 함수를 이용해 기존(수정 되기전) 데이터 영역에 대한 읽기 연산을 수행 중인 Reader가 남아 있는지를 체크한 후, 없을 경우 메모리 해제를 수행한다[3].

```
void reader() {
  for(;;) {
    rcu_register_thread();
    rcu_read_lock();
    x=rcu_dereference(gt);
    ...
    rcu_read_unlock();
    rcu_unregister_thread();
  }
}
```

```
void writer() {
  for(;;) {
    new=malloc(...);
    new->contents=new
value;
    old=rcu_xchg_pointer(&gt,
new);
    synchronize_rcu();
    free(old);
  }
}
```

표 1 URCU 예제 프로그램

2.3 Critical Section Routine

Critical section routine은 자원을 동시에 차지하기 위해 경쟁하는 두 개 이상의 프로그램 문제에 접근하는 방법이다. 두 개의 프로그램들이 동일한 카운터의 값을 증가시키려는 시도를 한다고 가정했을 때, 만약 두 프로그램 모두 동시에 변수를 가져와서 그 값을 증가시킨 뒤, 증가된 값을 다시 저장하는 일을 수행한다면, 증가된 값 들 중 하나는 없어지게 될 것이다. 오늘날의 프로세서들에서, 프로그램들은 fetch-and-op나, compare-and-swap(CAS)과 같은 read-modify-write 원자 명령어를 사용할 수 있다. 초창기의 프로세서들에서 이러한 명령어들은 존재하지 않았으며, 그 문제는 오직 평범한 어셈블러 명령어를 사용하여 원자적인 값의 증가를 수행하는 것이었다. 이 문제는 Edsgar Dijkstra에 의해 정의되었으며, 그에 의해 처음 해결되었다. "Critical section routine"은 바로 그가 문제를 풀었던 코드에 붙였던 이름이다[4].

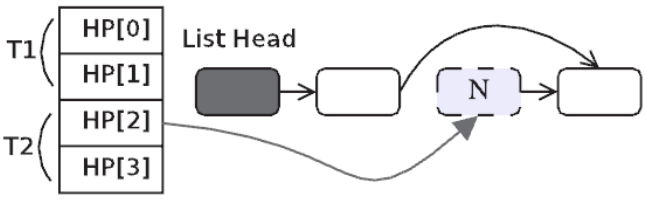

그림 4 HPBR 사용 예

3. 메모리 해제 기술

3.1 Hazard Pointer Based Reclamation(HPBR)

HPBR은 Hazard pointer를 기반으로 동적으로 할당되는 메모리에 대해 Lock-free한 메모리 해제 기법을 제공한다. Hazard pointer는 2004년 IEEE Transactions on Parallel and Distributed Systems에서 IBM Thomas J. Watson 연구소의 Maged M. Michael에 의해 처음 제안되었다.

Hazard pointer의 기본 메커니즘은 스레드가 자신의 메모리 사용을 다른 스레드에게 알려주는 것이다. 각각의 Reader 스레드는 자신만의 Single-writer/Multi-reader용 포인터를 가지고 있다. "Hazard pointer"라 명명된 이러한 포인터는 Reader 스레드가 공유 데이터에 대한 맵(map)을 자신의 Hazard pointer에 할당하면, 다른 스레드(writer)들이 해당 맵을 다른 맵으로 대체시키는 것은 허용하지만, 맵의 내용에 대한 수정이나 삭제는 불허한다. Writer 스레드는 대체된 맵을 삭제하기 전에 반드시 Reader의 Hazard pointer를 체크해야 된다. 만약 하나 이상의 Reader가 맵의 Hazard pointer를 사용 중이라고 한다면, Writer는 이 Hazard pointer의 사용이 끝날 때까지 맵을 삭제하지 않는다. HPBR에서는 Writer 스레드가 맵을 대체시킬 때마다 기존 맵의 포인터를 리스트에 보관한다. 삭제된 맵의 개수를 계산한 뒤에, 그중에서 Reader의 Hazard pointer를 조사한다. 만약 이 중에 Hazard pointer가 하나라도 없을 경우 삭제하는데 안전한 것으로 여기며, 만약 그렇지 않고 하나라도 Hazard pointer가 존재하면 다음 조사 때까지 해당 맵을 삭제하지 않는다.

위 그림 4는 HPBR의 사용 예를 보여주고 있다. 우측에 연회색으로 나타낸 노드 N은 링크드 리스트에서 이미 삭제되었지만, 스레드 T2의 Hazard pointer인 HP[2]에 의해 참조되고 있기 때문에 HPBR의 메커니즘에 의해 메모리 해제가 일어나지 않는다[5][6].

3.2 Lock-free Reference Counting(LFRC)

LFRC는 Lock-free한 참조 횟수(reference counting) 기반 가비지 컬렉션(garbage collection) 기법이다. LFRC는 일반적인 참조 횟수 기반 가비지 컬렉션 기법과 마찬가지로

할당된 각 오브젝트에 대한 참조 횟수를 기반으로 메모리 해제를 실시한다. 오브젝트에 대한 참조가 발생될 때마다 참조 카운터(reference counter)의 값이 증가하며, 참조 카운터의 값이 0이 될 경우 해당 오브젝트에 대한 메모리 해제를 수행한다. 하지만 이를 멀티스레드 환경에 적용할 경우 참조 카운터에 대한 병렬처리가 필요하며, Locking의 적용에 따른 추가적인 오버헤드가 발생하게 된다는 단점이 있다. LFRC는 참조 카운터에 대해 Lock-free한 접근을 구현함으로써 기존 기법의 성능을 향상시켰다[7][8].

3.3 ABA 문제

ABA 문제는 병렬화 기법 연구에서 자주 등장하는 현상으로, CAS(compare-and-swap) 연산 수행 시 공유 데이터의 변경 사항을 감지하지 못하는 현상을 말한다. 이는 CAS 연산에서 메모리 주소를 참조하던 도중 해당 메모리가 재사용되는 경우에 발생한다. 메모리 재사용의 경우 거의 모든 메모리 관리자가 사용하는 최적화 기법이기 때문에 병렬처리나 Lock-free 알고리즘 구현시 반드시 고려되어야하는 사항이다. 아래 표 2는 Lock-free 스택 구현에서의 ABA 문제의 예를 보여준다.

```
class Stack
{
  volatile Obj* top_ptr;

  Obj* Pop()
  {
    while(1)
    {
      Obj* ret_ptr = top_ptr;                  //(1)
      if (!ret_ptr) return NULL;               //(2)
      Obj* next_ptr = ret_ptr->next;           //(3)
      if(CompareAndExchange
            (top_ptr, ret_ptr, next_ptr))      //(4)
        return ret_ptr;
    }
  }

  void Push( Obj* obj_ptr )
  {
    ...
}
```

표 2 Lock-free 스택 구현에서의 ABA 문제

표 2의 코드를 병렬처리를 고려하지 않는 관점에서 바라보면 아무런 문제가 없어 보인다. 하지만 병렬처리를 고려하게 될 경우, 다음과 같은 상황이 발생할 수 있다. 스레드 A가 코드를 (3)까지 수행한 상태라고 가정했을 때, 로컬 변수인 ret_ptr과 next_ptr에는 이미 값이 저장되어 있는 상태이다. 여기서 next_ptr은 top_ptr이 변경되었을 경우 폐기되어야 하는 값이며, 이는 CAS 연산에 의해 처리된다. 그런데 스레드 A가 코드를 (3)까지 수행한 시점에서 스레드 B가 그 사이((3)과 (4) 사이)에 Pop()을 2회 수행한 다음, 다시 Push()를 1회 수행했을 경우, 2회의 Pop()에 의해 스택 상단의 2개 노드는 삭제된다. 그런 다음, 마지막 Push()에 의해 스택 상단에 저장된 노드의 값(스레드 A의 ret_ptr에 저장된 메모리 주소)을 메모리 관리자가 재사용하게 되면, 스레드 A가 실행을 재개했을 때 CAS 연산이 성공하게 된다. 폐기 되어야할 next_ptr의 값이 그대로 적용되어 버리는 것이다. 또한 현재 next_ptr에 저장되어있는 메모리 주소는 이미 폐기되어버린 상태이기 때문에 이후에 메모리 접근 위반이 발생하게 된다. 이와 같이 ABA Problem은 공유 데이터 처리 문제를 발생시키기 때문에, 병렬화 기법 연구에서 주요한 이슈로 다루어지고 있다[9].

4. 결론

본 논문에서는 Wait-free 동기화 및 메모리 해제 기법과 관련하여 주요 이슈 및 기술 현황에 대한 분석 결과에 대해서 기술하였다. Wait-free 동기화 기법은 Locking을 기반으로 하는 전통적인 동기화 기법보다 뛰어난 성능을 보여준다. 하지만 Wait-free 동기화 기법의 연구를 위해서는 Critical section routine, ABA 문제 등 병렬처리 관련 주요 이슈들 및 RCU, Hazard pointer 등과 같은 대표적인 기법들에 대한 보다 심도 있는 연구가 필요할 것이다.

참고문헌

[1] M. Herlihy, “Wait-Free Synchronization,” ACM Transactions on Programming Languages and Systems, 1991.

[2] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni, “Read-Copy Update,” In Ottawa Linux Symposium, 2001.

[3] M. Desnoyers, P. E. McKenney, A. Stern, M. R. Dagenais, and J. Walpole, “User-Level Implementations of Read-Copy Update,” IEEE Transactions on Parallel and Distributed Systems, to appear.

[4] Terms, “Critical Section Routine,” http://www.terms.co.kr/criticalsectionroutine.htm

[5] M. M. Michael, “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects,” IEEE Transactions on Parallel and Distributed Systems, 2004.

[6] Hart, T. E., McKenney, P. E., Brown, A. D., “Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation,” In Proc. 20th International Parallel and Distributed Processing Symposium, 2006.

[7] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele Jr., “Lock-Free Reference Counting,” In Proc. 20th Anniversary ACM Symposium Principles of Distributed Computing, 2001.

[8] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas, “Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting,” In Proc. 8th I-SPAN, 2005.

[9] Damian Dechev, Peter Pirkelbauer and Bjarne Stroustrup, “Lock-Free Dynamically Resizable Arrays,” Lecture Notes in Computer Science, 2006.